Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática

Ano 2013

António Manuel Malaquias de Almeida Valente

Implantação de sistemas operativos em módulos de comunicação sem fios

Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
Ano 2013

António Manuel Malaquias de Almeida Valente

Implantação de sistemas operativos em módulos de comunicação sem fios

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor José Alberto Gouveia Fonseca, Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e Mestre Paulo Jorge de Campos Bartolomeu, Diretor de I&D da Micro I/O Serviços de Eletrónica Lda

Trabalho apoiado pelo QREN no âmbito do Projeto “Concepção e Desenvolvimento de Sistema de Controlo de Estacionamento Inteligente”, SII&DT Mº30161, Janeiro 2013

COMPETE
PROGRAMA OPERACIONAL FACTORES DE COMPETITIVIDADE

QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL
PORTUGAL 2007.2013

UNIÃO EUROPEIA
Fundo Europeu de Desenvolvimento Regional

Dedico este trabalho aos meus pais, Manuel e Isabel, e à minha irmã, Ana.

o júri

Presidente

Professor Doutor Alexandre Manuel Moutela Nunes da Mota
Professor Associado do Departamento de Engenharia Eletrónica e Telecomunicações da Universidade de Aveiro

arguente

Doutora Fernanda Madureira Coutinho
Professora Adjunta, Instituto Superior de Engenharia de Coimbra

orientador

Professor Doutor José Alberto Gouveia Fonseca
Professor Associado do Departamento de Engenharia Electrónica e Telecomunicações da Universidade de Aveiro

Agradecimentos

Agradeço ao Professor Doutor José Alberto Fonseca pela orientação e pela paciência demonstrada no decurso deste trabalho, a si, o meu sincero obrigado.

Ao Mestre Paulo Bartolomeu, pela ajuda, pela boa vontade demonstrada e pela orientação prática deste trabalho assim como por todos os conhecimentos científicos que me tentou transmitir ao longo desta dissertação.

Aos meus restantes amigos, pela paciência, compreensão e amizade demonstradas ao longo destes últimos anos.

Aos meus pais agradeço por todo o apoio que me deram no decorrer não só da dissertação, mas em todo o meu percurso de vida, principalmente nos momentos mais difíceis.

À minha irmã, sem a qual a minha infância e a minha vida não seriam tão felizes.

À minha restante família, em particular aos meus avós e tios, por serem um pilar exemplo, com o qual muito me orgulho.

À Joana por tudo o que representa na minha vida.

Um bem-haja a todos.

palavras-chave Sistema operativo Contiki,Wireless Sensor Networsks,IPv6

resumo

A crescente adoção de redes de sensores sem fios na indústria tem potenciado esforços contínuos de investigação e desenvolvimento ao nível dos sistemas operativos que as suportam. A uniformização da programação para diferentes plataformas de *hardware* e a reutilização de pilhas de comunicação permite obter ganhos significativos no processo de desenvolvimento.

A Micro I/O possui uma plataforma de *hardware* designada por *µ*MRF que tem sido usada em várias aplicações de redes de sensores sem fios para validação de conceito e para prototipagem. Com vista a suportar aplicações com requisitos de comunicação mais exigentes e empregar um método de desenvolvimento mais eficiente, optou-se por adaptar o sistema operativo Contiki para a plataforma *µ*MRF.

Esta dissertação descreve o trabalho realizado para integrar o suporte do microcontrolador Microchip™ dsPIC33FJ256MC710 no sistema operativo Contiki. A validação deste trabalho foi realizada através de uma aplicação desenvolvida com a versão modificada do sistema operativo Contiki e com o sensor de temperatura existente na placa *µ*MRF. Esta aplicação permite o acesso remoto à temperatura do sensor, através de um *browser*, usando o protocolo HTTP e a verificação de conectividade usando pacotes ICMP, ambos sobre Ipv6. O desempenho da rede sem fios foi avaliado nas vertentes de perda de pacotes e de *round trip delay* usando pacotes *echo request* e *echo reply*.

O trabalho realizado insere-se num projecto para desenvolvimento de sistemas de “smart-parking” no qual as restrições de energia são significativas mas onde a possibilidade de aplicações, suportadas em sistemas operativos, é aliciante.

keywords Operating System Contiki,Wireless Sensor Networsks,IPv6,HTTP

Abstract

Over the last few years, there has been a significant increase in the use of wireless sensor networks in industry. This has pushed the research and development in embedded operating systems to support applications in this domain. In fact, the possibility to turn applications independent of the *hardware* platforms and the possibility to reuse communications stacks leads to significant gains in the applications development process.

Micro I / O has developed a *hardware* platform called *µ*MRF that has been used to build several applications with wireless sensor networks for proof of concept and prototyping. In order to support applications with more demanding communication requirements, and to improve the development process, it was decided to adapt the Contiki operating system to the platform *µ*MRF.

This dissertation describes the work done to adapt the Contiki operating system to support the Microchip ™ dsPIC33FJ256MC710 microprocessor. The validation of this work was conducted through an application developed with the modified version of the Contiki operating system using the temperature sensor onboard of the *µ*MRF platform for demonstration purposes. The developed application allowed remote access to the temperature sensor through a browser using the HTTP protocol and the connectivity verification, using ICMP packets, both through IPv6. The performance of the wireless network was evaluated through an analysis of the packet loss and round trip delays using echo request and echo reply.

This work is aimed to evaluate the possibility to support smart parking applications in embedded systems connected by wireless communications, with severe restrictions in power consumption but with a significant interest in having an embedded operating system with all the correspondent functionalities.

ÍNDICE GERAL

Índice de tabelas

Índice de figuras

Lista de abreviaturas e símbolos

ADC	*Analog-to-digital converter*
API	*Application Programming Interface*
BSD	*Berkeley Software Distribution*
Bit	*BInary digiT*
CAN	*Controller Area Network*
FIFO	*First In First Out*
HTTP	*Hypertext Transfer Protocol*
HTML	*HyperText Markup Language*
IoT	*Internet of Things*
IETF	*Internet Engineering Task Force*
6LoWPAN	*IPv6 over Low Power WPAN*
ICMP	*Internet Control Message Protocol*
LED	*Light-emitting diode*
LPRC	*Low power RC oscillator*
MAC	*Media access control address*
PLOSS	*Packet Loss*
PCB	*Printed Circuit board*
QoS	*Quality of service*
RTT	*Round-Trip Time*
SO	*Sistema operativo*
SPI	*Serial Peripheral Interface*
SLIP	*Serial Line Internet Protocol*
TCP/IP	*Transmission Control Protocol / Internet Protocol*
UART	*Universal asynchrounous receiver/transmitter*
USB	*Universal Serial Bus*
WSN	*Wireless sensor network*

1. Introdução

O desenvolvimento tecnológico e o surgimento de múltiplos componentes eletrónicos exigiram, desde sempre, o estudo e desenvolvimento de técnicas que permitissem a interação entre esses mesmos componentes e o ambiente envolvente, criando, consequentemente, valor para o Homem e para a sociedade. Adicionalmente, muitos componentes não apresentam qualquer utilidade quando usados individualmente sendo essencial a sua interligação em plataformas, de forma a maximizar as suas funcionalidades.

A eletrónica de instrumentação permite então a interligação entre três diferentes realidades: a eletrónica, o homem e o ambiente envolvente.

Associado ao desenvolvimento da eletrónica de instrumentação, tem-se verificado um aumento das capacidades computacionais apresentadas e aparelhos cada vez mais miniaturizados o que, aliado à descida dos seus preços, tem impulsionado o aumento da utilização de equipamentos eletrónicos em diversas áreas de investigação, de indústria e de lazer.

Esta evolução levou a que surgisse interesse em conceber uma rede composta por plataformas, com sensores integrados, capazes de obter informações sobre o meio que as rodeia, de processar essa mesma informação e de enviar os resultados processados ao utilizador, surgindo então o conceito de *Wireless Sensor Networks* (WSN).

Uma WSN é geralmente constituída por um vasto número de pequenos sensores integrados num único chip, usualmente chamadas nós ou "motes", com capacidades de comunicação, processamento e armazenamento de dados, que são autónomos ao nível da energia. Tipicamente, os nós da rede obtêm informações sobre o meio que os rodeia, através dos seus sensores integrados, pré processam-nos e transportam-nos através da rede até ao utilizador. Na Figura 1.1 é apresentada a arquitetura típica de um nó de uma WSN.

A investigação e desenvolvimento em WSNs foi inicialmente motivada por razões militares, uma vez que através de pequenos chips, passava a ser possível monitorizar vastas áreas geográficas, controlar de forma rápida e impercetível possíveis inimigos.Atualmente , o conceito de WSN é mais abrangente, sendo

possível encontrar WSN em várias indústrias, na monitorização de processos de fabrico, em estudos ambientais, através da verificação da qualidade do ar, ou mesmo para deteções prematuras de incêndios e de desastres naturais [1, 2].

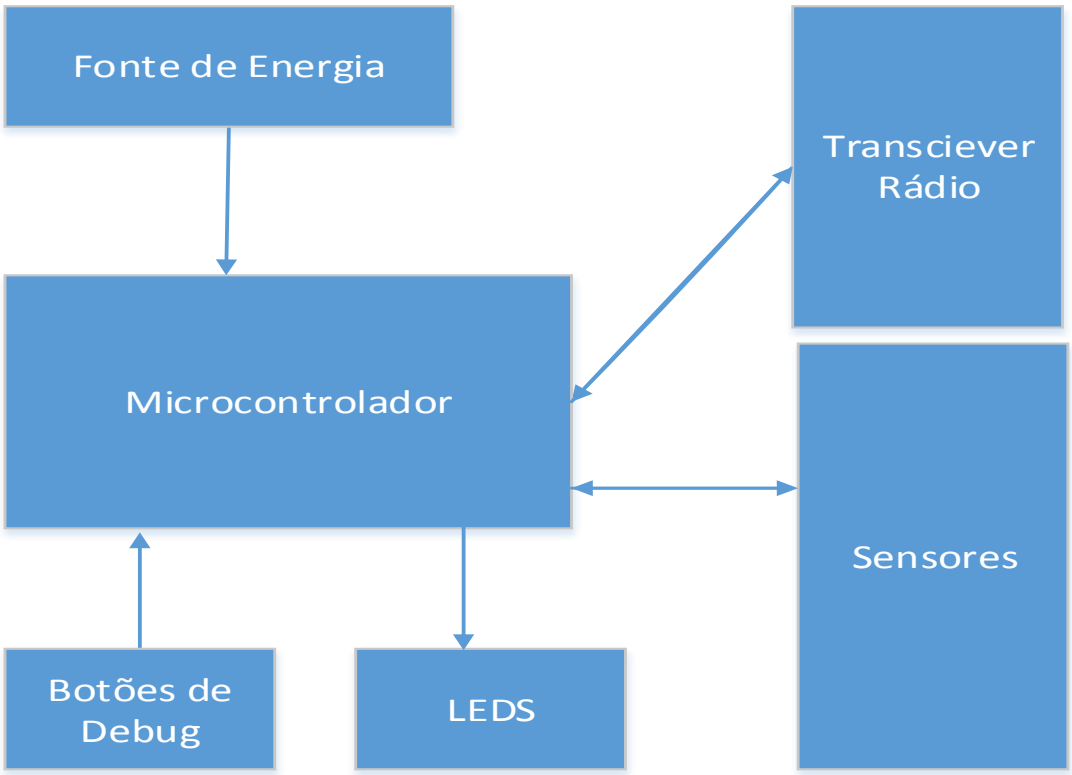

Figura 1.1: Arquitetura típica de um nó de uma WSN.

À medida que a eletrónica tem vindo a evoluir, esta passou a permitir que as WSNs fossem criadas em locais de cada vez mais difícil acesso, permitindo ao Homem obter informações de locais potencialmente hostis aumentando desta forma o seu interesse para a Ciência.

Atualmente, o desenvolvimento de aplicações para WSNs tem sido feito com o auxílio se sistemas operativos embebidos. De forma a evitar o conhecimento das características baixo nível das plataformas, e com vista a não impor um estudo prévio das mesmas, aquando do desenvolvimento de novas aplicações, tem-se tentado desenvolver sistemas operativos que imponham estas abstrações de alto nível, sendo esta opção também vantajosa na imposição de portabilidade de aplicações entre diferentes plataformas [3].

Os sistemas operativos (SOs) tradicionais para computadores pessoais, como por exemplo o SO Windows 7, da Microsoft, foram desenhados para sistemas onde os recursos de memória e de processamento ultrapassam em larga escala os que usualmente podem ser encontrados nas plataformas desenhadas para a criação de WSNs, pelo que a utilização destes SOs neste tipo de sistemas teve de ser repensada. Surgiu então o conceito de sistemas operativos para sistemas embebidos, com especial foco para os que possuíam capacidades de comunicação para desenvolvimento de WSN.

Tendo em conta as limitações que este tipo de plataformas tem a nível de memória, de processamento e energético, tentou-se criar uma camada de abstração entre os detalhes de baixo nível das plataformas e as futuras aplicações desenvolvidas para as mesmas.

Esta abstração forneceu aos engenheiros uma interface de gestão de processos, de interrupções, de memória, de periféricos, de suporte de rede e de políticas de escalonamento de tarefas, assim como uma total abstração do *hardware* existente nas diversas plataformas. Permitiu que os programadores focassem o seu trabalho somente no desenvolvimento de novas funcionalidades nas aplicações, abstraindo-se do *hardware* existente nas plataformas, reduzindo assim exponencialmente o tempo de desenvolvimento e o seu custo final.

A motivação desta dissertação passou então por:

- Fornecer à comunidade um documento que explica as principais linhas de funcionamento do SO Contiki.
- Conseguir obter uma nova transposição do SO Contiki para uma nova plataforma de desenvolvimento de soluções baseadas em WSNs.
- Mostras a robustez da transposição obtida para a nova plataforma, através do desenvolvimento de aplicações exemplo que usufruem das principais livrarias disponibilizadas pelo SO Contiki.

1.1. Organização do documento

No capítulo 1 é feita uma contextualização do campo do conhecimento no qual a corrente dissertação se insere, as motivações que levaram ao seu desenvolvimento e os objetivos propostos neste trabalho.

O capítulo 2 é dedicado à apresentação dos principais sistemas operativos para *Wireless Sensor Networks*, fazendo-se uma descrição dos seus prós e contras, prosseguindo-se com uma comparação dos diferentes sistemas operativos.

No capítulo 3 é descrito o sistema operativo Contiki de uma forma mais pormenorizada, evidenciando-se as suas características de funcionamento peculiares e as vantagens que essas características trouxeram ao ramo dos sistemas operativos para as *Wireless Sensor Networks*.

A implementação do sistema operativo numa nova plataforma é apresentada no capítulo 4, sendo feita uma descrição pormenorizada dos passos essenciais para que o sistema operativo Contiki seja transposto. Após a transposição do sistema operativo para a nova plataforma, é desenvolvida uma rede sensorial implementada com endereços IPv6 para fins de teste.

No capítulo 5 são efetuados testes de conectividade aos nós sensoriais pertencentes à rede sensorial de teste criada no final do capítulo 4 para validação do trabalho apresentado, havendo espaço para uma breve discussão sobre os resultados obtidos nas demonstrações efetuadas.

Por fim, no capítulo 6 são apresentadas as conclusões do trabalho desenvolvido e algumas sugestões para futuros trabalhos.

2. Sistemas Operativos para Wireless Sensor Networks

Nos dias de hoje cada vez mais se verifica uma procura significativa por novas tecnologias e por novas descobertas em todos os ramos da indústria e da investigação e é esta visão que tem impulsionado o desenvolvimento de sistemas eletrónicos. Atualmente já é possível criar redes com pequenos dispositivos eletrónicos, que, para além das suas capacidades de comunicação, têm intrínsecos a si pequenos sensores que, respondendo ao ambiente envolvente em que se encontram, conseguem dimensionar um vasto número de características que facilitam, ao Homem, o estudo do meio. Esta crescente evolução da eletrónica tem apresentado novos desafios, sendo o desenvolvimento de sistemas operativos para este novo tipo de plataformas, com vista à criação de redes sensoriais, o passo seguinte dessa evolução.

2.1. Introdução

O desenvolvimento de sistemas operativos embebidos com capacidades de comunicação para desenvolvimento de WSNs levantou desde a sua génese problemas que obrigou a uma evolução distinta do que até então acontecia no desenvolvimento de sistemas operativos para computadores pessoais.

Ao considerar-se um SO para sistemas embutidos com vista à criação de uma WSN, há questões sobre o seu desenvolvimento que impõem especial atenção. Um dos principais problemas encontrados prende-se com a crescente miniaturização das plataformas e, consequentemente, com as limitações em termos de recursos de processamento, memória e energia, que esta impõe.

As limitações encontradas nestes sistemas determinaram que, neste tipo de sistemas operativos, tivessem de ser desenvolvidos mecanismos leves e abstrações que permitissem apresentar um sistema capaz de fornecer serviços e funcionalidades para o desenvolvimento de novas aplicações sem exigir mais recursos que os disponíveis nas plataformas para as quais estavam a ser desenvolvidos.

2.1.1. Arquitetura

As soluções usualmente encontradas em sistemas operativos para WSN passam por arquiteturas monolíticas onde, associado ao *core* do SO, são ligadas todas as aplicações desenvolvidas, formando uma única imagem do sistema, ou, em alternativa, uma arquitetura modular onde existe uma separação bem delineada entre o *core* do SO, usualmente chamado de *kernel,* e os serviços necessários pelas aplicações que são associados à imagem do *core* mediante a sua necessidade.

Relativamente ao modelo de execução de tarefas é comum encontrar um modelo de execução baseado em eventos, sem preempção, isto é, para uma dada tarefa requerer o uso do microcontrolador esta terá que gerar um evento e, somente quando gestor de eventos der ordem, esta iniciará, mantendo o controlo do microcontrolador, até a sua execução ser terminada.

Este modelo de execução pode, no entanto, tornar-se problemático se a tarefa que controla o microcontrolador usufruir desse controlo durante um largo período de tempo, uma vez que levará a que outras tarefas, eventualmente mais prioritárias, estejam em espera, fenómenos usualmente conhecido por *starvation*. Observa-se, neste modelo de execução, a implementação de tarefas através de máquinas de estado, cujos estados são simples e rápidos o suficiente para que o fenómeno de *starvation* de tarefas não ocorra.

A arquitetura de SO baseada em *threads* é também encontrada habitualmente neste tipo de sistemas operativos. Neste modelo, verifica-se preempção da execução de *threads* por outras mais prioritárias, facilitando desta forma a implementação de aplicações concorrentes entre si. Como consequência, verifica-se, a cada troca de *threads*, aumento das mudanças de contexto de memória no controlo do sistema, impondo, desta forma, um aumento da carga computacional.

Têm existido vários debates sobre as vantagens do primeiro *versus* a do segundo, existindo até sistemas operativos que já adotam um modelo híbrido que combina as vantagens destes dois tipos de modelo, como é o caso do SO Contiki [4-6].

2.1.2. Gestão de memória

A gestão de memória é um dos pontos que requer especial atenção no desenvolvimento de sistemas operativos para sistemas embebidos. De facto, estes devem oferecer mecanismos de gestão de memória eficientes de forma a não sobrecarregar a pouca memória disponível.

Nos sistemas operativos tradicionais, por questões de segurança e proteção de dados, é prática comum a alocação de espaços de memória exclusivos para cada processo, levando a que, à medida que o número de processos aumenta, a memória necessária também aumente. Esta abordagem gera também um aumento da cópia de dados, e troca de contexto de memória entre processos, o que pode tornar-se insustentável para a eficiência das aplicações desenvolvidas.

Consultando o *Website* dos principais construtores de microcontroladores verifica-se que a atual geração de microcontroladores integra componentes que têm por norma cerca de 256 Kbytes de memória[7-9], tornando, desta forma, a gestão de memória um dos grandes desafios na conceção deste tipo de sistemas operativos.

A implementação e desenvolvimento de sistemas operativos para sistemas embebidos, com vista ao desenvolvimento de WSN, teve de chegar a um compromisso entre a memória disponível na plataforma e a *Quality of Service* (QoS) fornecida aos utilizadores da mesma. Este compromisso levou a admitir-se que a melhor forma de não sobrecarregar o microcontrolador passava por se admitir uma partilha de memória entre processos, conseguindo-se também, desta forma, uma redução significativa das necessidades de memória, em detrimento da segurança e proteção de dados.

2.1.3. Eficiência energética

Com a evolução da eletrónica têm surgido dispositivos cada vez mais pequenos e eficientes, levando a que hoje em dia já existam dispositivos militares impercetíveis aos nossos olhos, que fornecem informações cruciais, em tempo real, sobre o mundo que os rodeia.

Um dos problemas da miniaturização das plataformas tem a ver com o facto de a variação do tamanho e custo das plataformas estar proporcionalmente ligada à quantidade de energia disponível. Quanto mais miniaturizada for a plataforma,

menor será a energia disponível. Uma vez que a energia neste tipo de sensores é maioritariamente fornecida por baterias que, infelizmente, não têm tido o ritmo de desenvolvimento a que se tem assistido na eletrónica, pode o custo dos componentes variar de acordo com as especificações pretendidas e os seus graus de eficiência [10].

Uma gestão eficiente do consumo energético é, assim, um passo decisivo para o ciclo de vida de uma WSN, pois sendo a energia nestas plataformas um bem normalmente não renovável, uma má gestão do seu uso leva a um fim prematuro da WSN formada.

Neste tipo de redes, um dos principais consumidores de energia é, sem dúvida, a comunicação sem fios, pelo que é importante que o SO forneça mecanismos de otimização de energia, como é, por exemplo, o caso de colocar, periodicamente, a plataforma num estado de baixo consumo (*Sleep Mode)*, ativando a mesma quando necessário e de forma a não afetar as funcionalidades da mesma.

2.1.4. Portabilidade, Reprogramação e *Update*

A evolução que se tem verificado nos últimos anos no ramo da eletrónica de instrumentação tem permitido um aumento exponencial do *hardware* e funcionalidades disponíveis para desenvolver plataformas para WSN. Esta evolução da eletrónica tem desenvolvido microcontroladores mais eficientes e sensores mais fiáveis o que, aliado à redução do preço de acesso a estes novos dispositivos, gerou um *Boom* de novas plataformas para WSN.

Esta nova realidade levou a que os novos sistemas operativos para WSN tenham de ser de fácil portabilidade entre diferentes plataformas e diferentes módulos de *hardware*, propiciando assim o acesso a este novo mercado de componentes e facilitando a conceção de aplicações com outras funcionalidades.

A reprogramação de aplicações, ou mesmo a sua eliminação, deve ser também um processo de fácil gestão. Sabendo que uma WSN é muitas vezes composta por dezenas, ou até centenas de nós, e que muitas vezes estes ficam inacessíveis após a implementação da rede, a reprogramação em *run-time* da rede, através de comunicação sem fios, tem sido uma constante preocupação nos últimos tempos.

2.1.5. Consumo de Memória

Como referido anteriormente, as plataformas habitualmente usadas para criar WSNs são normalmente de tamanho muito reduzido e, consequentemente, limitadas em termos de recursos disponíveis. Uma das limitações deste tipo de dispositivos é, sem dúvida, a memória disponível, pelo que é de extrema importância que o SO implementado ocupe o menor espaço possível, permitindo assim que a memória remanescente fique disponível para as aplicações desenvolvidas para a rede.

2.1.6. Garantias de Tempo-Real

Aplicações de monitorização e de vigilância são de extrema importância, sendo um exemplo comum de uma tarefa periódica onde é necessário que haja requisitos de tempo-real a ser cumpridos. Existem outras aplicações que requerem que as suas tarefas sejam satisfeitas em tempo real, pelo que a implementação de um escalonador de tempo real é por vezes importante para o desenvolvimento de aplicações para WSNs com este tipo de requisitos.

2.1.7. Fiabilidade

A maioria das aplicações em WSNs é desenvolvida para que estejam em constante funcionamento durante um largo período de tempo, sendo de extrema importância que o SO contribua para o correto funcionamento das mesmas, permitindo, desta forma, o desenvolvimento de WSNs cada vez mais eficientes e complexas.

2.2. Sistemas operativos

Ao longo dos anos têm surgido vários sistemas operativos para sistemas embebidos com vista ao desenvolvimento de WSNs, sendo claramente o TinyOS [11] e o SO Contiki [6] os que maior atenção têm despertado de acordo com o *Google PageRank*™ da Figura 2.1. O *Google PageRank*™ utiliza uma família de algoritmos de análise de rede que atribui um peso numérico a cada página da Internet, medindo a importância que cada página tem da rede, sendo que, um *PageRank* maior corresponde a uma página de maior importância na rede.

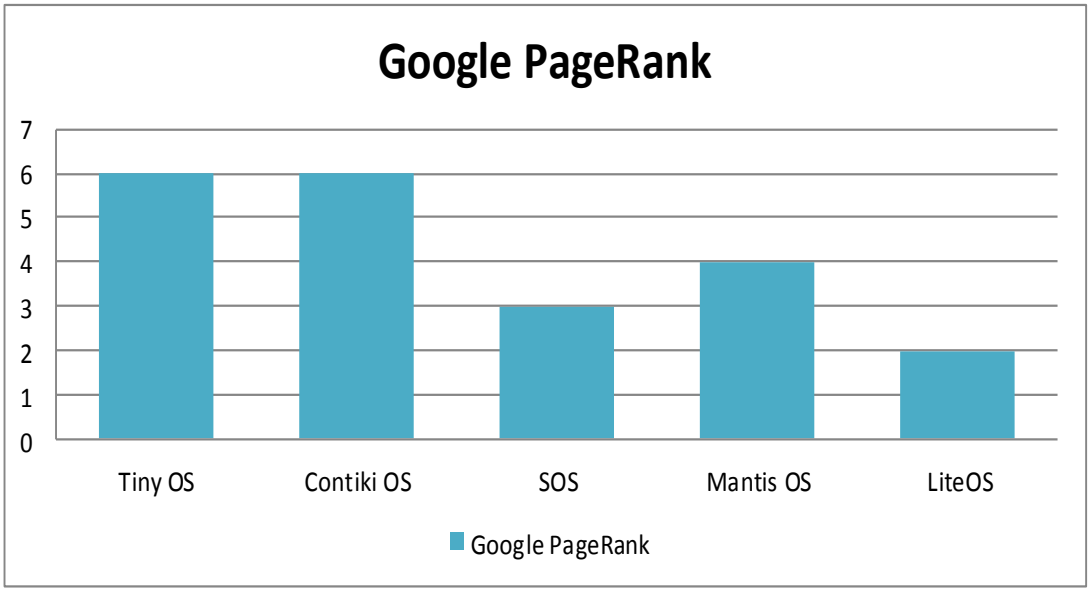

Figura 2.1: *Google PageRank* obtido a 8/01/2013.

2.2.1. TinyOS

TinyOS é um SO com baixo consumo de memória, desenvolvido na UC Berkeley, baseado em eventos (*event-driven*) para aplicações em rede *wireless* com uma arquitetura baseada em componentes (CBSE-*Component-Based software Engineering*), fornecendo um sistema de comunicação baseado em mensagens ativas (AM), e distribuído através de uma licença *open source Berkeley Software Distribution* (BSD) [11].

A utilização de um modelo baseado em eventos suporta aplicações concorrentes, usando uma quantidade reduzida de memória, aumentando desta forma

o rendimento em comparação com sistemas baseados em *threads*, uma vez que pode rapidamente criar tarefas associadas a eventos, sem qualquer bloqueio ou *polling.*

As tarefas no SO são escalonadas através de uma fila do tipo *First in First Out* (FIFO), permitindo ao processador entrar em *sleep mode* quando a mesma se encontra vazia, mantendo os sensores periféricos em funcionamento mas reduzindo substancialmente o consumo de energia por parte da plataforma.

O SO é um sistema monolítico, pelo que os programadores têm que alocar todos os recursos necessários para o SO e aplicações durante o seu desenvolvimento, não podendo alterar as aplicações em *run-time*.

A reprogramação de um nó da rede pode ser feita através do mecanismo de reprogramação Deluge [12] para TinyOS [11], sendo que, devido às suas características monolíticas, as aplicações e o *kernel* têm de ser transpostos para o nó numa única imagem.

As aplicações para TinyOS são escritas em nesC [13], uma otimização da linguagem C feita exclusivamente para *WSN* de memória limitada, necessitando, portanto, de uma maior curva de aprendizagem para o desenvolvimento de aplicações.

2.2.2. ContikiOS

ContikiOS foi desenvolvido por Adam Dunkels, no Swedish Institute of Computer Science, e é um sistema operativo para sistemas embutidos, tais são os nós de uma WSN de código livre sobre uma licença *open source*, baseada na 3ª cláusula do BSD. A sua configuração básica ocupa menos do 10 Kbytes de RAM e 30 Kbytes de ROM, e fornece três *comunication stacks* (μIP [14], μIPv6 [15] e Rime [16]) assim como funcionalidades *multi-threading* [6].

Este SO, assim como todos os seus módulos e aplicações, desenvolvido em linguagem C, facilita o desenvolvimento de novas aplicações, uma vez que esta linguagem é dominante nos projetos de sistemas embutidos.

Tipicamente, o sistema operativo Contiki consiste num *kernel*, em livrarias, num *program loader* e num conjunto de processos inicializadores do sistema. As comunicações entre processos efetuam-se sempre através do *kernel*, que, por sua vez, não fornece uma camada de abstração de *hardware*, deixando que as aplicações e

hardware comuniquem entre si livremente. Um processo é definido por um bloco de controlo e pelo *Process Thread*, que é usualmente chamado de *event handler*, devido às suas funções de gestão dos eventos recebidos pelo processo. Este é mantido em memória, sendo que o *kernel* guarda somente um ponteiro para bloco de controlo do processo. Todos os processos partilham o mesmo espaço de memória e não correm em diferentes domínios de proteção.

Observando o SO Contiki de outra perspetiva, pode-se dividir o sistema operativo em duas componentes: o *core* e as aplicações do utilizador. O *core* é compilado numa única imagem binária e normalmente não é modificado após o seu desenvolvimento. O *program loader* fica encarregue de carregar e descarregar as aplicações do utilizador no sistema durante a sua execução, quer através de uma *comunication stack*, quer diretamente na memória física (por exemplo EEPROM), sendo considerado por isso um sistema modular [17].

2.2.3. SOS

SOS é um SO desenvolvido para WSN pelo *Networked and Embedded Systems Lab (NESL)* na *University of California* de *Los Angeles* [18]. É um SO baseado em eventos e adota uma arquitetura modular, isto é, o SO consiste num *kernel* estático e num conjunto de módulos que podem ser dinamicamente associados ao *kernel* [19].

O escalonador é parte integrante do *kernel* e opera através de uma FIFO com duas prioridades. A atualização das aplicações, à semelhança do SO Contiki [6], não requer a substituição integral da imagem do SO, o que torna a reprogramação dos módulos da rede mais eficaz energeticamente.

Os seus criadores deram por finalizado o desenvolvimento deste sistema operativo, deixando, no entanto, o seu código disponível para desenvolvimentos do mesmo dentro da sua comunidade *on-line*.

2.2.4. Mantis OS

O SO Mantis foi um dos primeiros sistemas operativos baseado em *threads* para WSNs, apresentando-se com uma licença *open source* BSD que impõe poucas restrições no que diz respeito à edição e distribuição do código fonte do sistema [20].

Os arquitetos deste SO acreditaram que um modelo baseado em *threads* era o que melhor respondia às necessidades de concorrência entre aplicações de WSN. Este SO foi também concebido de modo a ter *multi-threads*, que seriam preemptivamente escalonadas, facilitando desta forma a implementação de aplicações para WSNs cada vez mais complexas.

De forma a ter uma curva de aprendizagem relativamente rápida, as aplicações para este sistema operativo são escritas através da linguagem de programação C.

2.2.5. LiteOS

O LiteOS é um dos mais recentes SOs de tempo real para *Wireless Sensor Networks*. Foi desenvolvido na *University of Illinois* e é distribuído também ao abrigo de uma licença *open source* BSD. Foi desenhado para fornecer ao utilizador um ambiente semelhante ao oferecido pelo UNIX [21].

Inclui um sistema hierárquico de ficheiros e uma interface *wireless* para interface com utilizador, baseada em comandos UNIX. Assente numa arquitetura modular, o *kernel* suporta carregamentos dinâmicos e tem execução *multi-threading* nativa, assim como *debugging* online. O LiteOS suporta *updates* de *software* entre o *kernel* e as aplicações do utilizador, que estão ligadas através de *system calls*.

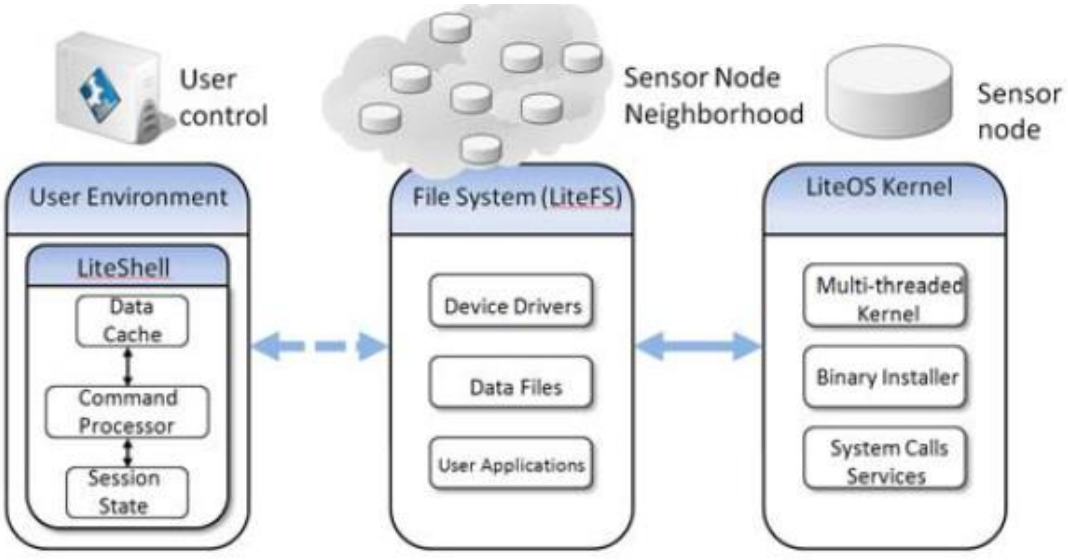

Figura 2.2: Arquitetura do SO LiteOS [21].

Podemos observar, na Figura 2.2, que a arquitetura do LiteOS está dividida em três subsistemas: LiteShell, LiteFS e o Kernel.

O LiteShell está implementado num *personal computer* (PC) e fornece ao utilizador comandos Unix para interagir com os nós da rede *wireless* na sua presença.

A interface LiteFS fornece suporte a operações quer sobre ficheiros quer sobre diretórios.

O *kernel* aplica uma política de *threads*, deixando, no entanto, que haja aplicações que controlem eventos usando funções *callback* para maior eficiência. Este SO implementa um escalonamento, baseado em prioridades e *round-robin*, e suporta carregamentos/descarregamentos dinâmicos das aplicações, assim como um conjunto de *system calls* para separação entre *kernel* e aplicações [17].

No desenvolvimento de aplicações, usa-se a linguagem Lite++,uma linguagem moderna orientada para objetos, considerada mais *user-friendly* que C ou nesC [21].

2.3. Comparação dos sistemas operativos apresentados

No subcapítulo anterior, foram apresentados os principais sistemas operativos para sistemas embebidos, de código aberto, e com vista ao desenvolvimento de WSN. Apesar de alguns adotarem mecanismos de execução semelhantes, os serviços e funcionalidades disponibilizados são diferentes.

Esta secção sumaria as principais vantagens e desvantagens dos sistemas operativos anteriormente apresentados, seguindo-se a escolha do sistema operativo a transpor.

Tabela 2.1: Análise das características dos SO para WSN.

Características	TinyOS	Contiki	SOS	Mantis OS	Lite OS
Arquitetura	Monolítico	Modular	Modular	Modular	Modular
Modelo de Execução	Eventos	Hibrido	Hibrido	Threads	Hibrido
Escalonamento por prioridades	✗	✓	✓	✓	✓
Escalonamento Real-Time	✗	✗	✗	✓	✓
Reprogramação dinâmica	✓	✓	✓	✗	✓
Linguagem suportada	nesC	C	C	C	LiteC+
Networking	Active Message	µIP, µIPv6, Rime	Message	“comm”	File-assisted

Analisando a Tabela 2.1 e a Figura 2.1 podemos observar que os sistemas operativos que têm despertado maior atenção nos últimos tempos foram o TinyOS e o SO Contiki. Desses, o sistema operativo Contiki é o que tem despertado maior interesse na indústria, graças à sua implementação da pilha de comunicação IP, que devido ao seu tamanho reduzido veio revolucionar o conceito de redes sensoriais aproximando-as do conceito da *Internet of Things* (IoT), onde cada nó pertencente às redes sensoriais passaria a ser identificado por identificadores IP únicos, numa estrutura semelhante à da Internet.

Associado à pilha IP do sistema operativo Contiki, existe um módulo que implementa uma solução, desenvolvida pelo grupo de trabalho do *Internet Engineering Task Force* (IETF) IPv6 over Low power WPAN (6LowPAN) [22]. Este módulo veio resolver a questão de como implementar pacotes IPv6 sobre redes baseadas na norma IEEE 802.15.4 o que, aliado aos pedidos da indústria para disponibilizar a plataforma μMRF, desenvolvida pela Micro I/O Serviços de Electrónica Lda, com um sistema operativo, fez recair o trabalho desta dissertação no desenvolvimento de uma transposição do sistema operativo Contiki para a plataforma μMRF.

3. Sistema Operativo Contiki

Neste capítulo é fornecida ao leitor uma caracterização da arquitetura do SO Contiki assim como uma perspetiva do seu *modus operandi*. De seguida, é feita uma exposição das principais funcionalidades disponíveis no SO Contiki para o desenvolvimento de aplicações para WSNs.

Após leitura deste capítulo é esperado que o leitor esteja munido do conhecimento necessário para desenvolver aplicações para o sistema operativo assim como seja detentor de uma perceção de quais os passos necessários para se obter uma solução que vise a implantação do sistema operativo numa nova plataforma.

3.1. Caracterização

O sistema operativo Contiki, desenvolvido por Adam Dunkels, no *Swedish Institute of Computer Science,* tem um modelo de execução que se considera híbrido, sendo uma mistura das principais vantagens encontradas no modelo baseado em *threads* e no modelo de execução baseado em eventos.

No desenvolvimento de sistemas operativos para sistemas embebidos, com vista à sua utilização em WSNs, um dos principais problemas encontrados é a escassez de recursos de memória que as plataformas têm ao seu dispor. Nestes ambientes de escassez de recursos tem havido ao longo dos anos uma discussão acesa sobre as vantagens e desvantagens entre escolher o modelo tradicional de *multi-threading* ou o modelo baseado em eventos para a obtenção da concorrência necessária para o desenvolvimento de serviços e funcionalidades cada vez mais complexas [5, 17].

Se por um lado o sistema de execução baseado em *threads* facilita a vida dos programadores ao permitir a existência de mecanismos de bloqueio de *threads*, em detrimento da execução de outras de prioridade superior, este modelo de execução apresenta também uma desvantagem que pode limitar a sua utilização no mundo das WSNs.

A desvantagem deste modelo é a imposição de que cada *thread* tenha uma região de memória exclusiva o que, aliado à dificuldade de prever as reais

necessidades de memória de cada *thread*, leva a que os recursos reservados sejam muitas vezes majorados, levando assim a que, à medida que as aplicações aumentam a sua complexidade, as necessidades de memória aumentem, podendo tornar-se insustentáveis para este tipo de plataformas.

As crescentes necessidades de memória que existem no modelo de execução baseado em *threads* são solucionadas se se optar por um modelo de execução baseado em eventos. Este segundo modelo é capaz de fornecer concorrência sem que exista necessidade de haver a obrigatoriedade de haver uma reserva de espaço de memória exclusiva por *thread.*

No modelo de execução baseados em eventos, os processos têm associado a si *event-handlers* que ficam responsáveis por fornecer respostas aos eventos a eles destinados. Os processos são ativados na receção de um evento e uma vez escalonados não admitem preempção. Devido a esta característica todos os processos podem usar a mesma região de memória, reduzindo desta forma a quantidade de memória necessária [6].

Apesar de promissor, este último modelo também apresenta algumas desvantagens, maioritariamente devido à impossibilidade de um processo sofrer preempção, levando a que processos com execuções longas monopolizem o uso dos recursos do microcontrolador. Uma vez que este modelo de execução não permite o uso de abstrações de bloqueio, a maioria das aplicações são desenvolvidas através de uma máquina de estados que controla o fluxo de funcionamento, levando a que a programação de novas aplicações apresente um custo de desenvolvimento elevado [6, 23, 24].

O SO Contiki surge em 2002 neste panorama de baixos recursos disponíveis e de dois possíveis modelos de execução, tendo o seu criador, Adam Dunkels, inovado criando uma solução híbrida, tentando obter uma solução que alcançasse o melhor dos dois modelos.

Apesar de ser considerado como tendo um modelo híbrido o SO Contiki recai maioritariamente na categoria dos sistemas operativos baseados em eventos, apresentando, contudo, características que facilmente o distinguem e que podem ser encontradas no modelo baseado em *threads*, graças à utilização de uma abstração que introduz condições de bloqueio condicionais, as *protothreads.*

Em sistemas puramente *event-driven*, as aplicações têm de ser tipicamente implementadas através de máquinas de estado. No SO Contiki isso deixa de ser necessário com o uso de *protothreads*. Esta nova abordagem apresenta vantagens em relação a sistemas *multi-threading* uma vez que as *protothreads* não impõem uma reserva de memória por *thread*. Ao contrário das *threads,* uma *protothread* é somente uma função que não interfere com a execução de outras, assim como não pode impor bloqueios dentro de funções chamadas pela mesma. Igualmente, em contraste com o modelo baseado em *threads*, o uso *da*s *protothreads* permite que se possa impor bloqueios explícitos, facilitando que possa ser feita uma leitura sequencial das aplicações por parte dos programadores, assim como uma maior perceção de quais funções podem ser bloqueadas [24].

Esta nova abstração terá direito, mais tarde na subsecção 3.1.2, em *Protothreads*, a uma explicação mais detalhada no que diz respeito ao seu funcionamento e implementação.

No SO Contiki, uma aplicação consiste num ou mais processos, que têm associados a si *event-handlers*, responsáveis por gerirem as respostas aos eventos direcionados aos processos à semelhança do que acontecia num sistema operativo baseado puramente em eventos. De cada vez que um evento é destinado a um processo, o evento é escalonado, sendo entregue ao *event-handler* do processo destino que tomará o controlo do microcontrolador. Os parâmetros recebidos pelo *event-handler* permitem que, para além de receber o evento, seja também possível receber dados e usá-los, se necessário, no processamento subsequente.

Uma característica associada também ao SO Contiki é o facto de o seu *kernel* não apresentar mecanismos de proteção de dados associados o que significa que qualquer aplicação pode aceder e corromper qualquer informação existente no sistema operativo. Esta abordagem, apesar de pouco ortodoxa, é a que garante o menor consumo de memória, pois evita a reserva de espaços de memória entre processos, sendo este um dos compromissos que Adam Dunkels teve de assumir. Outra razão, para a existência desta partilha de memória entre os *kernel* e as aplicações, prende-se com a eficiência do uso do processador, evitando-se assim trocas de contexto de memória encontrados em sistemas baseados em *threads*, que

tanto ocupam o tempo de processamento do microcontrolador atrasando, por consequência, as aplicações que nele são executadas.

Esta partilha de memória é também interessante uma vez que as aplicações podem partilhar dados entre si, e com o *kernel*, através do uso de variáveis globais. Apesar de esta partilha interligar entre si aplicações e sistema operativo, esta metodologia de partilha de dados torna o uso da memória disponível extremamente eficaz, sendo que, as imagens típicas deste sistema operativo ocupam menos de 10 Kbytes de RAM e 30 Kbytes de ROM [25].

3.1.1. Kernel

O *kernel* é o coração do sistema operativo Contiki, encontrando-se definido no ficheiro "process.c", presente no directório "./core/sys/" do sistema operativo. Este módulo contém as API's necessárias à manipulação de processos, de eventos, e é também o responsável pelo escalonamento dos processos. Neste módulo existem dois objetos principais: a lista de processos ativos e a fila de eventos assíncronos do tipo FIFO.

A lista de processos ativos é implementada como uma lista ligada de estruturas de processos. Esta lista contém somente os processos que já foram inicializados e que se mantêm ativos no sistema. Os processos podem ter sido ativados no arranque do sistema operativo, ou inicializados mais tarde por outros processos.

O tamanho da lista de processos varia dinamicamente ao longo da execução das diferentes aplicações, uma vez que estas são livres de ativar e terminar processos.

A fila de eventos assíncronos, por seu lado, está implementada através de um *buffer* circular, cujo tamanho está definido na variável constante PROCESS_CONF_NUMEVENTS e pode ser alterado aquando da compilação do sistema operativo.

Os eventos assíncronos são direcionados para esta fila aquando da sua criação, sendo que, como já foi referido previamente, os eventos síncronos não passam por esta fila, sendo imediatamente entregues ao(s) processo(s) destino. Este *buffer* mantém os eventos que ainda não foram invocados pelo escalonador do sistema operativo, ficando os eventos em espera.

3.1.1. Eventos

No SO Contiki, existem dois tipos de eventos: eventos assíncronos e eventos síncronos. Os eventos assíncronos são colocados na fila de eventos assíncronos do *kernel*, e mais tarde enviados aos seus processos destino quando o escalonador no sistema operativo ordena o envio do evento para o(s) processo(s) destino. Esta fila de eventos assíncronos do *kernel* é do tipo FIFO, pelo que os eventos são escalonados pela ordem com que os mesmos chegaram à fila.

Tabela 3.1: API de envio de eventos do OS Contiki.

Função	Descrição
process_post()	**Envio de evento assíncrono para processo(s)**
process_post_synch()	**Envio de evento síncrono para processo(s)**
process_poll()	**Ativa pedido de poll para processo**

Quando um evento é entregue ao processo destino, é invocado o *event-handler* do mesmo com um argumento, especificando o evento que recebeu.

O *kernel*, por outro lado, gera os eventos síncronos imediatamente, fazendo com que o(s) processo(s) recetor(s) seja(m) imediatamente escalonado(s) e o seu(s) *event-handler*(s) invocado(s). Depois de enviado o evento e entregue ao(s) processo(s) destino, o processo que enviou o evento manter-se-á bloqueado até o processo(s) recetor(es) do evento gerado terminar(em) a(s) sua(s) execução(ões), altura em que o processo original retomará a sua execução [24].

Outra característica interessante nos eventos é o facto de estes poderem estar destinados a um único processo destino ou, em alternativa, a todos os processos

ativos no sistema. A atribuição do processo destino dos eventos é efetuada durante o funcionamento do sistema operativo de forma dinâmica, quando a API de envio de eventos do Contiki injeta o evento no sistema. Um evento é iniciado ou pelo *kernel* do sistema operativo ou por um processo ativo no sistema.

Os *device drivers* dos elementos existentes nas plataformas são implementados muitas vezes de forma a imporem um contexto preemptivo, através do uso de interrupções para despoletar as respetivas rotinas de serviço à interrupção o que, no caso do sistema operativo Contiki, pode gerar *race conditions* entre essas mesmas rotinas de serviço à interrupção e os processos escalonados. No SO Contiki foi imposto um método diferente para a implementação dos *device-drivers*. Nas diferentes rotinas de interrupção dos *device drivers*, em vez de serem processados os dados recebidos pelos diferentes dispositivos, estes irão somente fazer um pedido de *poll* para um processo, ficando este responsável pelo posterior processamento do que anteriormente era feito dentro das rotinas de serviço à interrupção. Funciona assim cooperativamente com os restantes processos do sistema operativo. Este mecanismo de *poll* será discutido mais à frente no subcapítulo 3.1.2, em *Polling* de processos.

O *kernel* só é responsável por criar e enviar eventos para um processo nas seguintes situações: na criação do processo onde lhe envia o evento INIT; na terminação de processos onde envia os eventos EXIT e EXITED; e quando é feito um pedido de *poll* a um processo, enviando ao mesmo um evento de POLL. Os processos, por seu lado, podem enviar eventos indiscriminadamente entre si.

Os *event-handlers* associados aos processos são os únicos recetores de eventos em todo o sistema operativo, sendo eles os únicos responsáveis pelo processamento dos mesmos.

A estrutura adotada pelo SO Contiki para os eventos está especificada no ficheiro “process.c”, encontrado no diretório do sistema operativo “/core/sys/” e pode ser observada na Tabela 3.2.

Tabela 3.2: Estrutura de eventos Contiki.

Campo	Descrição
ev	**Número do evento**
Data	**Ponteiro para dados associados ao evento**
p	**Ponteiro para o(s) processo(s) recetor**

O campo "ev" contém o valor numérico associado ao evento. Cada tipo de evento, como por exemplo o INIT, EXIT e EXITED, têm associado a si um valor único no sistema, que serve para o *kernel* e os diferentes processos conseguirem distinguir e fornecer respostas diferentes para diferentes tipos de eventos. Este campo será entregue ao *event-handler* do processo especificado no campo "p", o processo destino e recetor do evento, quando o evento é escalonado.

Adicionalmente pode também ser associado ao evento, no campo "Data", um ponteiro para uma variável, para ser também entregue ao processo.

É importante realçar que devido à partilha de memória existente entre *kernel* e os processos existentes no SO Contiki, é possível partilhar dados simplesmente através do uso de ponteiros, de processo para processo, evitando assim que os dados tenham de ser copiados do espaço de memória do processo emissor para o espaço de memória do processo recetor do evento.

O campo "p", como tinha sido referido anteriormente, especifica o processo destino do evento pelo que, caso seja destinado a um único processo, corresponderá a um ponteiro para a estrutura do dito processo. Caso o evento esteja destinado a todos os processos ativos no sistema, este campo terá uma constante especial, PROCESS_BROADCAST, que está definido no *kernel* do SO Contiki com o valor NULL.

3.1.2. Processos

No SO Contiki um processo é composto por duas componentes fundamentais: o bloco de controlo e a *Process Thread* que é responsável por ser *event-handler* do processo [24]. É através do bloco de controlo que o *kernel* do sistema operativo consegue manter o registo de todos os processos ativos para que estes possam ser escalonados quando chegam eventos a eles direcionados, ou restaurado o seu estado, quando estes deixam de estar bloqueados.

Por cada processo na lista de processos ativos, seis campos de informação são guardados, como especificado na estrutura de processos do *kernel*, e ilustrado na Tabela 3.3.

Tabela 3.3: Estrutura de um processo em Contiki.

Campo	Descrição
Next	**Ponteiro para o próximo processo da lista de processos ativos**
Name	**Nome do processo**
Thread	**Ponteiro para o *event-handler* do processo**
Pt	***Protothread* do processo**
state	**Estado do processo (None;Running,Called)**
needspoll	**Sinalização para verificar processo (pedido de POLL)**

Um processo no SO Contiki pode assumir três estados distintos, sendo que, os diferentes processos podem assumir diferentes estados ao longo da sua existência no sistema. O estado do processo é mantido no campo "state" da estrutura do processo.

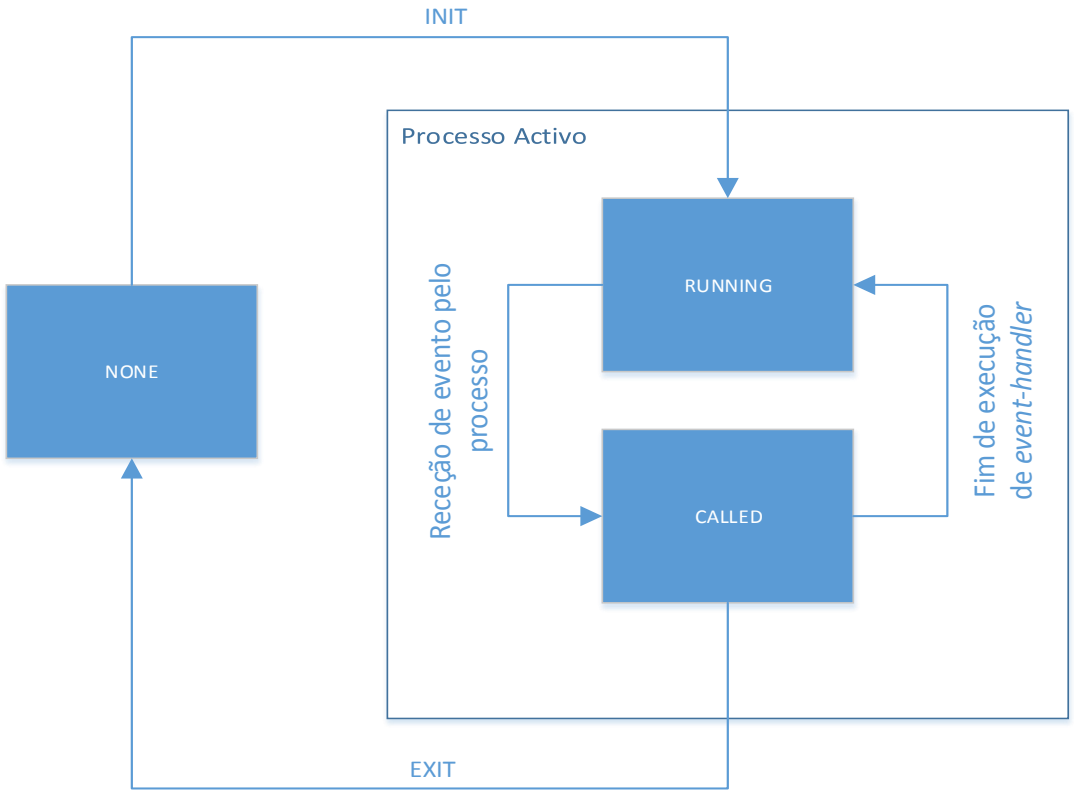

Figura 3.1: Fluxograma dos estados do processo.

Na Figura 3.1 apresenta-se um fluxograma dos diferentes estados possíveis de um processo e quais as suas possíveis transições durante a sua execução no SO Contiki.

Assim que o processo é inicializado, este é adicionado à lista de processos ativos do *kernel*. O estado do processo transita para *RUNNING* e é enviado um evento síncrono INIT para o mesmo, para que o *event-handler*, associado ao processo, tenha a oportunidade de executar o código de inicialização do processo. Quando o processo é chamado pelo *kernel*, através do escalonamento de um evento direcionado ao mesmo, este transita o seu estado para *CALLED*, mantendo este estado enquanto o seu *event-handler* estiver em execução.

Quando um processo é terminado, um evento síncrono do tipo EXITED é enviado a todos os processos ativos, com exceção do processo a terminar, para que os recursos associados ao processo cessante possam ser reclamados pelos outros processos. Um evento do tipo EXIT será posteriormente enviado para o processo a terminar para que o mesmo possa, através do seu *event-handler*, ser terminado.

Finalmente o campo "needspoll" é uma variável que é usada para sinalizar os processos que necessitam de ser verificados. Esta sinalização está intimamente associada aos *device drivers*, uma vez que estes estão usualmente associados ao uso de interrupções e, tendo uma interrupção caráter preemptivo, esta pode gerar problemas de *racing* entre o processo ativo, no momento em que surge a interrupção e o consequente processamento de informação, efetuado dentro rotina de serviço à interrupção gerada.

De forma a evitar a situação referida anteriormente, foi criado no sistema operativo este método de sinalização de processos que garante o envio de um evento para os processos cujo campo "needspoll" esteja ativo por parte do escalonador do sistema. Normalmente estes processos ficam associados ao processamento de informação recebida ou por enviar para o *device driver*, deixando a rotina de interrupção associada ao *device driver* de estar responsável por esse mesmo processamento, passando só a ativar o campo "needspoll" do processo que, de forma cooperativa com os restantes processos ativos no sistema, irá fazer o processamento necessário. Este método é vulgarmente chamado de *polling* no SO Contiki e será mais tarde discutido no subcapítulo 3.1.2, em *Polling* de processos.

Protothreads

Em sistemas embutidos com baixos recursos de memória, a solução que impunha um consumo mais eficiente da memória era, até agora, a dos sistemas baseados em eventos. Contudo este modelo apresenta uma grande desvantagem para os programadores de aplicações: as tarefas, uma vez em execução, não podem ser bloqueadas por outras, tendo como consequência ter que se implementar as aplicações sob a forma de máquinas de estado, com o seu fluxo de controlo bem

definido, pois não existe hipótese de bloqueá-lo, tornando esta tarefa complexa e morosa para os programadores.

As *protothreads* surgem então na implementação do SO Contiki para que se possa usufruir da eficiência do uso de memória presente nos sistemas com modelos de execução baseados em eventos e adicionar uma funcionalidade que até então somente estava presente nos sistemas baseados em *threads*, os bloqueios de tarefas, mas desta vez sem a necessidade de reserva de memória por *thread* [6, 24].

Com a introdução de bloqueios condicionais, a programação de aplicações deixa de ser obrigatoriamente feita com base em máquinas de estado, facilitando assim a sua implementação.

Como foi exposto anteriormente, as aplicações desenvolvidas para o SO Contiki são baseadas em processos que, por sua vez, são constituídos por um bloco de controlo e um *event-handler*. O *event-handler* associado a cada processo está implementado através de uma *protothread*, tendo sido esta implementação a maneira encontrada para que os processos possam ter a sua execução bloqueada. A utilização de *protothreads* é totalmente imperceptível para o programador de aplicações, uma vez que este somente utilizará a API de gestão processos, que se encontra exposta na Tabela 3.4.

O bloqueio de processos no SO Contiki é feito através da macro PROCESS_WAIT_UNTIL(). Esta leva consigo uma condição que garantirá o bloqueio da *protothread* do processo até que a mesma deixe de ser satisfeita.

Uma vez bloqueado o processo, a condição que deu direito ao seu bloqueio será novamente avaliada a cada invocação do processo, sendo que, caso a condição deixe de ser satisfeita, a *protothread* do processo reiniciará após a macro que deu origem ao seu bloqueio. Em caso contrário, o processo manter-se-á com a sua *protothread* bloqueada.

A implementação das *protothreads* encontrada no SO Contiki não salvaguarda a memória destas entre bloqueios, sendo que, a solução encontrada para que as variáveis locais dos processos sejam salvaguardadas passa por defini-las como variáveis “static”, uma vez que, desta forma, estas são guardadas fora do domínio de memória correspondente à *protothread* e, consequentemente, não serão apagadas [23].

Tabela 3.4: API de gestão de processos do SO Contiki.

Função	Descrição
PROCESS_BEGIN()	**Define o início de um processo**
PROCESS_END()	**Define o fim de um processo**
PROCESS_WAIT_EVENT() ou PROCESS_YIELD()	**Espera que um evento seja enviado para o processo**
PROCESS_WAIT_EVENT_UNTIL (c) ou PROCESS_YIELD_UNTIL(c)	**Espera que um evento seja enviado para o processo com uma condição extra**
PROCESS_YIELD_UNTIL(c)	**Garante que o processo é bloqueado e cede o controlo do microcontrolador pelo menos uma vez até a condição imposta ocorrer**
PROCESS_WAIT_UNTIL(c)	**Semelhante à anterior, sem garantias de cedência de controlo do microcontrolador (deve ser usada com cuidado).**
PROCESS_EXIT()	**Sai do processo que está a ser executado**
PROCESS_PT_SPAWN(pt,thread)	**Cria uma novo processo dentro da *protothread* de um outro processo,**
PROCESS_PAUSE()	**Cede o controlo do microcontrolador, com uma condição de bloqueio válida só por uma vez**
PROCESS_POLLHANDLER(handler)	**Especifica uma ação para quando o processo receber um evento POLL**
PROCESS_EXITHANDLER(handler)	**Especifica uma ação para quando o processo terminar**

No SO Contiki, o início e o fim de uma *protothread* está delimitado pelas macros PROCESS_BEGIN() e PROCESS_END() respetivamente. A existência de código relativo à *protothread* fora destas macros delimitantes pode resultar em comportamentos do sistema inesperados [23, 24].

Uma *protothread* pode ser terminada prematuramente se, durante a sua execução, for usada a macro PROCESS_EXIT().

O bloqueio incondicional de um processo está também disponível através das macros PROCESS_WAIT_EVENT() ou PROCESS_YIELD(), verificando-se que, após o bloqueio, a *protothread* só retornará a sua execução quando o processo for novamente invocado pelo escalonador.

No SO Contiki, o estado da *protothread* do processo é guardado no bloco de controlo de processo, no campo "pt" definido na Tabela 3.3.

Quando a *protothread* do processo é bloqueada, o seu estado de execução tem que ser guardado pelo sistema operativo de forma a poder retornar à sua execução quando a condição de bloqueio deixar de estar satisfeita.

O mecanismo de salvaguarda do estado de execução da *protothread* do processo é conhecido no SO Contiki pelo nome de "continuação local". Este mecanismo é semelhante ao encontrado nos sistemas de *threads* para garantir a continuação das mesmas, sendo que, no mecanismo implementado no SO Contiki não é guardada memória reservada pois as *protothreads* não a têm.

Nas *protothreads*, o estado de execução é definido por um ponto de continuação na função que está a ser executada. A "continuação local" é baseada em duas operações possíveis: LC_SET e LC_RESUME. Quando acontece uma operação de LC_SET, a linha de código da execução é guardada, podendo deste modo ser mais tarde retornada a execução com a operação LC_RESUME.

O uso do mecanismo de salvaguarda do estado das *protothreads* é também ele imperceptível ao programador, estando o seu uso implícito nas macros da API de controlo de processos que encontramos na Tabela 3.4.

Polling de processos

Como tínhamos visto os processos no SO Contiki são escalonados através de eventos, sendo depois executados até que os seus *event-handlers* sejam bloqueados ou até que estes cheguem ao fim da sua execução. Este método de execução, em situações onde a execução de rotinas está dependente não de eventos mas de interrupções, como acontece usualmente nos *device drivers* onde ocorra entrada e saída de dados, não é viável.

Associado às rotinas de serviço à interrupção, e de forma a evitar *race conditions* com processos já escalonados, o sistema operativo Contiki desenvolveu uma técnica de executar o código que evita o processamento de dados de entrada e saída nessas rotinas de serviço à interrupção, fazendo-o mais tarde e de forma cooperativa com os processos ativos no sistema, através de um processo criado para esse efeito.

Na estrutura dos processos, presente na Tabela 3.3, podemos identificar um campo para sinalizar o processo especificamente criado para o *device-driver*. Assim, na rotina de interrupção, é somente feita a alteração do campo "needspoll" desse processo, através da API " Request Process to be polled" existente no sistema operativo, deixando o processo que estava a ser executado com o controlo do microcontrolador. Quando o escalonador do *kernel* for executado, este verificará se existem processos que tenham o campo "needspoll" ativo. Caso existam, este irá enviar um evento do tipo POLL ao processo criado em complemento ao *device-driver* o qual, através do seu *event handler*, irá executar o código necessário ao processamento dos dados recebidos ou a enviar.

Podemos pois afirmar que associados aos *device drivers* que funcionem por interrupções, estão associadas duas componentes fundamentais à sua correta execução: a rotina de serviço à interrupção, que ativará o campo "needspoll" da estrutura do processo, e a componente associada com o processamento da informação que se encontrará no *event-handler* associado ao processo.

Controlo de processos

O SO Contiki fornece aos seus utilizadores uma API para o controlo de processo, para que a gestão dos processos, eventos e respetivos escalonamentos seja feita de forma simples e eficaz.

Aquando da inicialização do sistema operativo é feita a inicialização de processos fundamentais ao correto funcionamento do sistema operativo na plataforma em questão. Estes processos, inicializados durante o arranque do sistema operativo, estão normalmente relacionados com a entrada e saída de dados, assim como serviços que sejam necessários como, por exemplo, os diferentes tipos de *timers*.

Os processos e serviços a ser inicializados no arranque do sistema operativo têm de ser especificados no ficheiro "main.c" característico de cada plataforma. Se um serviço não é necessário pode não ser incluído na lista de processos a inicializar de forma a reduzir o consumo de recursos de memória. No sistema operativo Contiki é também responsabilidade do *kernel* gerir o envio de eventos aos processos à medida que estes são criados e escalonados.

Os processos podem ser inicializados no arranque do sistema operativo ou durante a execução de processos já ativos. Uma vez ativos, os processos são livres de enviar eventos a outros processos, de inicializar novos processos assim como de desativar processos que se encontrem na lista ligada de processos do *kernel*.

Escalonamento

Os processos no SO Contiki são escalonados quando lhes é enviado um evento por parte do escalonador do *kernel* ou quando o campo de *"needspoll"* de um processo ativo estiver também ativo.

Quando o SO Contiki inicia, assim como quando um processo é bloqueado, o controlo retorna à função "main()" do SO que se encarrega de chamar o escalonador através do comando "process_run()", ficando o segundo responsável por determinar qual o processo que irá executar de seguida.

O escalonador do sistema operativo garante que os processos que tenham o campo “needspoll” ativo tenham prioridade sobre todos os outros processos, sendo que, se existir mais do que um processo com esse campo ativo, estes são despachados sequencialmente.

O escalonador do SO Contiki sabe da existência de processos com o campo “needspoll” ativo, por observação da variável global “poll_requested”, sendo que, nesse caso, irá efetuar o comando “do_poll” para escalonar o processo que necessita de ser executado. Caso a variável “poll_requested” esteja com o seu valor nulo, o escalonador irá escalonar o próximo evento na lista de eventos assíncronos do *kernel*, chamando o *event-handler* do processo recetor do evento. O funcionamento do escalonador pode ser seguido no fluxograma da Figura 3.2.

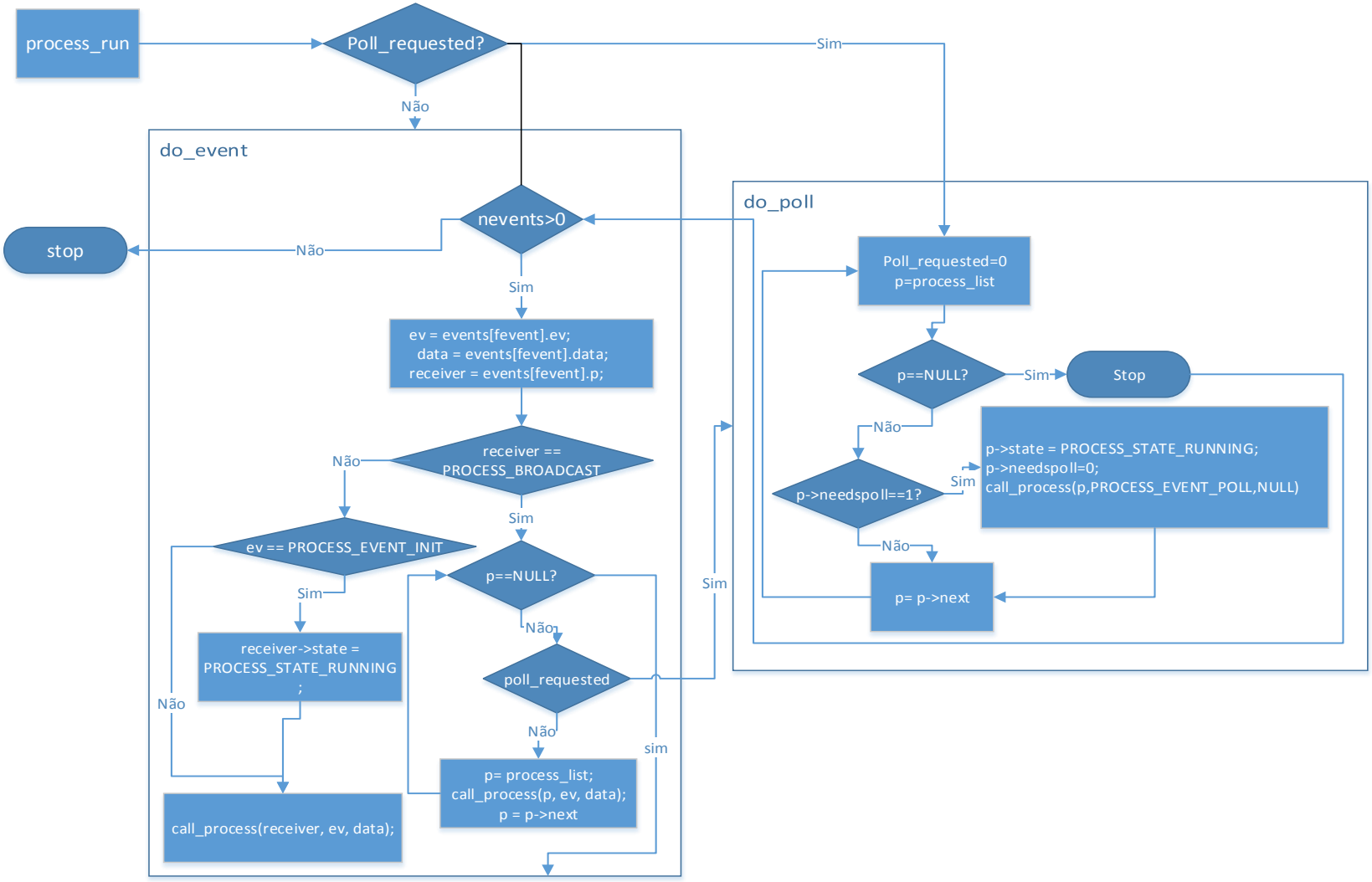

Figura 3.2: Escalonamento no SO Contiki.

3.1.3. Alocação de memória

O SO Contiki fornece aos programadores três soluções de gestão de memória: o *memory block allocator* (memb), o *managed memory allocator* (mmem) e a standard C malloc [26].

A livraria memb é a mais frequentemente usada. Nesta livraria, os blocos de memória são alocados num *array* de objetos de tamanho constante e colocados na memória estática. Na Figura 3.3, encontra-se um exemplo do funcionamento desta livraria.

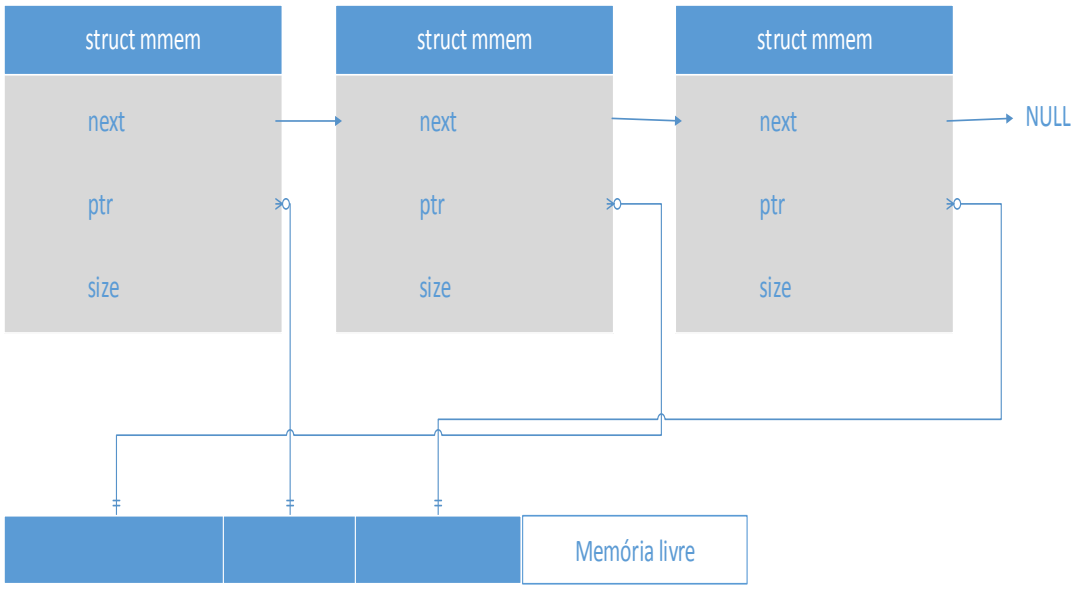

Figura 3.3: Memory block allocator [24].

A livraria mmem fornece um serviço de alocação semelhante ao mmaloc, fornecendo também um sistema de desfragmentação automática da área de memória gerida.

Em sistemas com pouca memória, as alocações de memória, efetuadas dinamicamente, podem originar fragmentações de memória, pelo que é comum adotarem-se mecanismos de alocação estáticos. É neste paradigma que funciona o mecanismo de alocação de memória *standard* C malloc.

A alocação e eliminação de objetos de diferentes tamanhos torna-se no entanto problemática, razão pela qual a utilização desta livraria não é a mais aconselhada.

3.1.4. Sistema de Ficheiros

Por razões económicas e energéticas, as plataformas normalmente utilizadas em WSNs apresentam-se com poucos Kbytes de RAM e de memória *Flash*. Em muitas aplicações existe a necessidade de o sistema ser capaz de armazenar dados no próprio nó da rede, ou porque o sistema está desconectado temporariamente da rede para uma maior poupança ou porque os dados só precisam de ser enviados após um breve pré-processamento no nó.

O sistema operativo Contiki fornece um sistema de ficheiros versátil, chamado COFFEE [27], que fornece as operações comuns de escrita e leitura sobre ficheiros armazenados na memória *Flash*.

A utilização de estruturas log tem ganho alguma relevância na gestão de memórias *flash*. Tipicamente, as memórias *Flash* apresentam uma limitação em relação às memórias baseadas em discos magnéticos no que diz respeito à reescrita de ficheiros. Nas memórias *flash* é impossível reescrever um ficheiro sem primeiro apagar uma grande secção de memória, pelo que, ao guardar as alterações de um ficheiro numa estrutura, ao invés de reescrever imediatamente na memória, aumenta a eficiência deste tipo de memórias [28].

Desenhado especialmente para memórias *Flash* e EEPROM, o sistema de ficheiros COFFEE criou um mecanismo semelhante às estruturas log, mais eficiente na utilização da memória RAM, e permitindo uma otimização da memória por ficheiro [27].

Quando um ficheiro é modificado, o COFFEE cria uma estrutura micro log, que é indexada ao ficheiro. O micro log difere das estruturas log usuais. Cada ficheiro aberto usa O(1) RAM, e permite a otimização por ficheiro. Quando um micro log enche, o COFFEE, de forma transparente, junta a informação do mesmo à *file* original num novo ficheiro, e apaga os 2 ficheiros anteriores [27].

Este sistema de ficheiros tem algumas desvantagens, como, por exemplo, a necessidade de definir o tamanho da estrutura micro log aquando da sua criação. Esta

situação leva a que, caso haja necessidade de armazenar mais informação do que o tamanho do micro log, seja criado um ficheiro maior e copiado o conteúdo do ficheiro original e do micro log para o novo ficheiro, levando a um maior atraso devido à cópia de ficheiros.

3.1.5. Dynamic Loader API

O SO Contiki oferece uma interface de reprogramação dinâmica que agregar novos módulos ou altera módulos já existentes no sistema operativo.

O formato ELF foi criado com objetivo de criar um padrão para os arquivos binários gerados pelos compiladores, tendo sido o escolhido para ser utilizado com a API *dynamic loader* do Contiki [29].

Um ficheiro ELF tem um cabeçalho seguido por secções de dados. O ficheiro tem as seguintes secções: código binário (.text), dados estaticamente alocados com valores pré-atribuídos (.data), dados não inicializados (.bss), tabela de símbolos (.symtab) e, por último, a secção (.strtab) onde são armazenadas as *strings* [30].

Para que o ficheiro seja aceite pelo *loader* do Contiki, este deve conter pelo menos a lista de secções descritas [30].

3.2. Visão do programador

A criação de aplicações para o SO Contiki, como já anteriormente foi referido, passa pela criação de processos que, utilizando os mecanismos de bloqueio associados às *protothreads*, tornam o sistema de execução, baseado em eventos do Contiki, mais próximo de um modelo *multi-thread*.

Deste modo, é importante perceber como é feita a criação de um processo no SO Contiki. Na Figura 3.4, apresenta-se um código exemplo para obtermos uma melhor perceção.

Na primeira linha, o bloco de controlo do processo é definido com o nome do processo que, neste exemplo, se designa por “example_process”, e um texto pelo qual o processo se identificará, sendo, neste caso, “Processo exemplo”.

O SO Contiki possui também um mecanismo, módulo *autostart*, onde os processos podem ser automaticamente iniciados quando o sistema inicia, ou quando

um módulo que contém processos é introduzido no sistema. O propósito deste mecanismo é fornecer ao programador a capacidade de informar o sistema, através de uma lista, sobre quais os processos ativos, assim como de terminar os processos que aparecem nessa mesma lista [24]. Assim, após a definição do processo, este é introduzido na lista de ponteiros de processos a ser iniciados na inicialização do sistema operativo, através da macro AUTOSTART_PROCESSES().

Após esta etapa, passamos à definição da *protothread* do processo. Esta inclui o nome do processo ("example_process") e duas variáveis para entrada para eventos: ev e data.

```
PROCESS(example_process,"Processo exemplo");
AUTOSTART_PROCESSES(&example_process);

PROCESS_THREAD(example_process, ev, data)
{
/*Variáveis locais*/
static int ptr;

 PROCESS_BEGIN();

while(1)
 {
  PROCESS_WAIT_UNTIL(ev == condição1);

 if(qualquer coisa)
  {
   /*...*/
   PROCESS_WAIT_UNTIL(ev == condição2),
   /*..*/
  }
}
PROCESS_END();
}
```

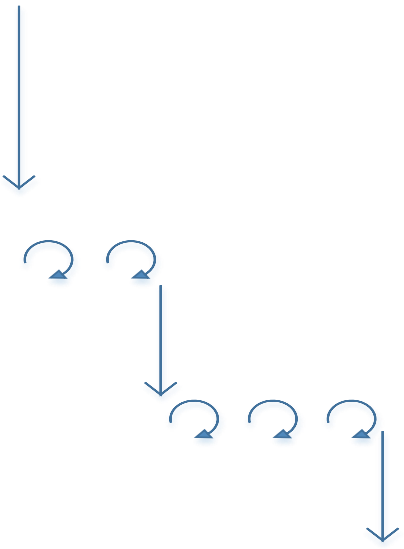

Figura 3.4: Exemplo de criação de um processo com *protothreads*.

A *protothread* é inicializada com a declaração da macro PROCESS_BEGIN(). O código colocado depois desta declaração será executado em cada invocação do processo.

A utilização da macro "PROCESS_WAIT_UNTIL(ev == condição)" irá devolver o controlo do microcontrolador ao *kernel*, bloqueando o processo, enquanto este não receber um evento igual à condição.

Quando o *kernel* escalona um evento para o processo, a condição que mantém o processo bloqueado é verificada, sendo que, caso esta já não seja verdadeira, o processo deixa o estado bloqueado, iniciando a sua execução após a macro que deu origem ao bloqueio. Caso contrário, o processo mantém-se em espera e o controlo do microcontrolador permanece no *kernel.*

Um processo em SO Contiki deve ter um "PROCESS_BEGIN()" e um PROCESS_END(). No exemplo da Figura 3.4, uma vez que o processo é mantido num *loop* infinito, o "PROCESS_END()" nunca é atingido. No entanto, a sua declaração é obrigatória, pois esta contém implícito o código necessário à correta implementação das *protothreads* no SO Contiki [23, 24].

3.2.1. Livrarias

O *kernel* do SO Contiki fornece somente um sistema de gestão de eventos, sendo o resto do sistema implementado sob a forma de livrarias que são opcionalmente agregadas à imagem do sistema, consoante a necessidade das aplicações desenvolvidas [6].

De seguida, vamos expor algumas das principais livrarias disponíveis para uso no sistema operativo Contiki.

Timers

O sistema operativo Contiki fornece um conjunto de livrarias *timer* que são usadas quer por aplicações quer pelo próprio sistema operativo. As livrarias *timer* são responsáveis por verificar se um dado temporizador já expirou, acordar o sistema do estado de baixo consumo energético e por escalonar tarefas *real-time*. As diferenças entre os diferentes módulos *timer* impõem diferentes utilizações dos mesmos: alguns permitem longas utilizações devido à sua baixa granularidade temporal e outros apresentam maior resolução temporal, mas com menor alcance, devido às limitações impostas pelo tamanho da variável que guarda o valor do timer [31].

As livrarias *timer* têm como base um módulo *clock* que é responsável por fornecer funcionalidades de gestão do tempo do sistema e também de bloqueio do microcontrolador durante um curto período de tempo.

As livrarias *timer* e *stimer* são as versões mais simples dos *timers* e são usadas para verificar se um dado período de tempo já passou. A diferença entre estes dois *timers* é a sua resolução de tempo: os *timers* usam os *ticks* do *clock* do sistema, enquanto os *stimers* usam segundos para permitir períodos de tempo muito mais longos. Ao contrário de outros *timers*, as livrarias *timer* e *stimer* podem ser usadas no contexto de uma interrupção, o que as torna especialmente úteis para *device drivers* de baixo nível.

A livraria *etimer* fornece *event timers* que geram eventos quando expiram, sendo usados para escalonar eventos para processos Contiki.

A livraria c*timer* fornece *callback timers* que são usados para escalonar chamadas para funções *callback* após um período de tempo. À semelhança dos *timers* de eventos, estas são usadas para esperar algum tempo enquanto o resto do sistema pode trabalhar ou entrar em modo de baixo consumo. Uma vez que os *callback timers* chamam uma função quando o *timer* expira, eles são especialmente úteis para qualquer código que não tenha um processo de Contiki explícito, como é muitas vezes o caso das implementações de protocolos. A livraria *rtimer* fornece escalonamento de tarefas *real time*.

A livraria *rtimer* preende qualquer processo do Contiki que esteja a correr de forma a permitir que as tarefas *real-time* sejam executadas no seu tempo [31].

Serial-Line API

O mecanismo de receção de dados via comunicação série está especificado na livraria *serial-line* disponibilizada pelo SO Contiki. Já a saída de dados através da porta série é implementada pela livraria *standard do C* [32].

O mecanismo de leitura da porta série é específico do sistema operativo e baseia-se na geração de uma interrupção quando um caráter da porta série está pronto para ser lido. Na rotina de serviço, à interrupção chama-se função de leitura do caráter, que fica responsável por armazenar os caracteres lidos até receber o caráter de fim de linha. Quando é lido o caráter fim de linha, é gerado um evento de *polling* para o *serial_line_process* que, quando ativo, envia para todos os processos ativos do sistema um evento, juntamente com um ponteiro para a *string* lida na porta série [32].

Esta livraria, para ser funcional, necessita que seja criado um ficheiro que a interligue com o *device-driver* associado à porta série do microcontrolador.

SLIP API

O sistema operativo Contiki disponibiliza nas suas livrarias um modo de funcionamento da porta série díspar do encontrado na livraria Serial-Line, e que se baseia na implementação do protocolo Serial Line IP definido no *standard* ITEF RFC 1055 [33].

O protocolo SLIP [33] é um protocolo que encapsula pacotes IP para posterior envio via porta série o que, apesar de já obsoleto, ainda é utilizado em WSNs, graças à sua vantagem de impor pouca sobrecarga computacional. Esta livraria, disponível no SO Contiki, necessita de ser interligada com o *device-driver* da "UART" do microcontrolador, para se tornar funcional.

LEDs API

A utilização de LEDs em plataformas com microcontroladores resume-se, normalmente, na alteração do estado do registo de saída atribuído ao LED para o estado '1', para ligar o led anexado a esse registo, e para o estado '0' para o desligar.

Apesar da simplicidade de programação neste tipo de plataformas, o SO Contiki implementa uma livraria que cobre todas as hipóteses de utilização dos LEDs da plataforma. Esta livraria torna-se importante uma vez que uniformiza a utilização dos LEDs, independentemente da plataforma que esteja a correr SO Contiki, acelerando assim a curva de aprendizagem para programar aplicações para este SO [32].

Esta API é inicializada através da função leds_init(), estando posteriormente disponível um vasto leque de funções que permitem usufruir de forma simples dos diferentes LEDs da plataforma [32].

Cada LED da plataforma tem um identificador que, por norma, está definido no *core* do SO Contiki, no ficheiro core\dev\leds.h, sendo o procedimento normal

definir o identificador do led através de macros do tipo "LEDS_xxxx", onde "xxxx" é, por norma, a cor do dito LED.

O vetor do estado dos leds é independente da plataforma uma vez que o mapeamento dos LEDs é, por norma, feito por *default* no core/dev/leds.h. Se a plataforma tiver LEDs diferentes dos padrão, estes terão que ser incluídos e definidas as suas macros.

A função "leds_toggle()" tem as mesmas funções da função "leds_invert()" e só foi mantida por razões de compatibilidade com versões anteriores [32].

Tabela 3.5: API de gestão dos LEDs da plataforma.

Função	Descrição
leds_init ()	**Inicia API de gestão de LEDs da plataforma**
leds_get ()	**Retorna um vetor com o estado atual dos LEDs**
leds_on (l)	**Liga os LEDs identificados por "l"**
leds_off(l)	**Desliga os LEDs identificados por "l"**
leds_toggle(l)	**Troca o estado dos LEDS identificados em "l"**
leds_invert(l)	**Semelhante ao comando anterior**

Comunicação

A criação de uma WSN só se torna possível se as plataformas tiverem a capacidade de formar uma rede, comunicando entre si as informações relativas aos seus sensores. Uma das vantagens da utilização de sistemas operativos para WSNs é a possibilidade de utilizar pilhas de comunicação já desenvolvidas para os mesmos, evitando assim que tenham de ser desenvolvidos de raiz os mecanismos de comunicação.

No SO Contiki, a comunicação foi implementada de forma a permitir múltiplas pilhas de comunicação. Estas comunicam com as aplicações através de eventos síncronos e é possível utilizar um *buffer* para todo o processamento da comunicação. Desta forma, não são efetuadas cópias de dados constantemente, otimizando-se o tempo utilizado.

Um *device-driver* do *transceiver* rádio é responsável por efetuar a leitura do pacote IEEE 802.15.4 recebido no *buffer* e posteriormente efetua a chamada da camada superior do serviço de comunicação. A pilha de comunicação processa os cabeçalhos do pacote e envia um evento síncrono para a aplicação onde o pacote está destinado. A aplicação opera posteriormente no pacote recebido e, opcionalmente, coloca uma mensagem no *buffer* antes de devolver o controlo à pilha de comunicação.

A pilha de comunicação é responsável por acrescentar o cabeçalho à mensagem da aplicação e devolver o controlo ao *device-driver* para que o pacote seja transmitido. O modelo de comunicação adotado no Contiki está esquematizado na Figura 3.5.

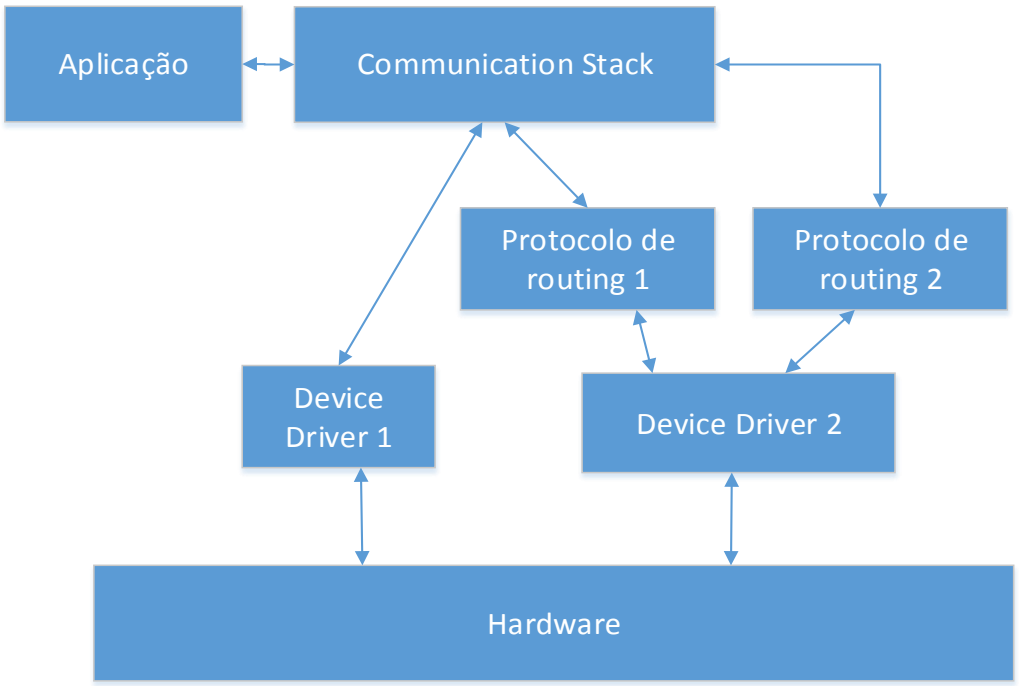

Figura 3.5: Modelo de Comunicação do SO Contiki [6].

O SO Contiki contém duas pilhas de comunicação: µIP e Rime. A µIP, desenvolvida numa parceria entre a Cisco, Atmel e SICS, é compatível com pilha TCP/IP tornando possível comunicar através da Internet. Esta implementa protocolos IPv4, IPv6, TCP e UDP (sendo os dois últimos compatíveis com os dois primeiros). De forma a ser compatível com a maioria das plataformas de uma WSN, o µIP foi um projeto de desenvolvimento que criou uma pilha de comunicação de dimensão reduzida.

A µIP necessita de uma camada inferior (de acordo com o modelo OSI) para comunicar com os seus pares, que podem ser nós da rede, ou um *gateway*.

Quando a comunicação é entre nós da rede, esta é feita via comunicação sem fios, se a versão da pilha de comunicação escolhida for a que implementa IPv6. O SO Contiki usa para gestão da camada MAC o módulo chamado *siclowmac*, que tem uma implementação do mecanismo de encapsulamento e de compressão dos cabeçalhos de pacotes IPv6 para pacotes IEEE 802.15.4. Implementação descrita no *standard* IETF RFC 6282 [34] e desenvolvida pelo grupo 6LowPAN, reencaminhando, posteriormente, o pacote IPv6 convertido para o *device-driver* do

transceiver rádio e vice-versa. Se a versão escolhida for IPv4, o Contiki escolhe um *routing* tipo *mesh*, feita através da pilha de comunicação Rime.

Quando a comunicação pretende ser para fora da WSN, é necessário comunicar com um *gateway*. Por norma é utilizado um PC, embora possa ser outro sistema embutido. A comunicação entre o PC e a plataforma da WSN, é por regra feita através de comunicação série. Os pacotes IP são enviados entre as duas plataformas, através do módulo Serial Line IP (SLIP) [33]. Do lado do computador, o programa deve fazer a interface entre a ligação série e a rede. Dependendo da versão μIP escolhida, as suas funcionalidades variam.

Com o μIPv6, o nó será carregado com uma aplicação simples, que ficará responsável por reencaminhar os pacotes IPv6 para a *serial line* e vice-versa, sendo o PC responsável por efetuar todo o trabalho protocolar remanescente, nomeadamente a descompressão de cabeçalhos 6lowPAN e junção de pacotes IPv6 fragmentados.

Se for usada a μIPv4, o processo de comunicação com a *gateway* funcionará de maneira diferente. O nó conectado ao PC irá atuar como *gateway*, com a pilha de IP implementada nele. Cada vez que o pacote tem de ser enviado, o nó verificará o seu endereço IP: Se pertencer à sua WSN, ele enviará o pacote através do seu *transceiver* rádio, caso contrário, enviará o pacote para o PC usando a ligação série. O PC irá então correr um programa que criará a interface para a rede IP.

O Rime por seu lado é uma *communication stack* de tamanho reduzido, desenhada para dispositivos de rádio de baixa potência. Os protocolos no Rime estão construídos segundo camadas, sendo os protocolos mais complexos implementados através de outros mais simples [35].

O Rime suporta primitivas de comunicação *single-hop* e *multi-hop*. De notar que as primitivas *multi-hop* não especificam como é que os pacotes são transportados através da rede. À medida que o pacote é transportado na rede, a aplicação, ou uma camada superior do protocolo, em cada nó, é invocada para escolher o próximo nó, permitindo desta forma implementar protocolos de *routing* arbitrários sobre as primitivas *multi-hop* [35].

4. Implantação em Módulos de Comunicação sem fios

Neste capítulo é feita a descrição da plataforma de desenvolvimento para a qual é pretendido fazer a transposição do SO Contiki, sendo apresentadas as ferramentas de desenvolvimento utilizadas no decurso deste projeto. De seguida, explica-se detalhadamente o processo de transposição do sistema operativo para uma nova plataforma, acabando o capítulo com uma implementação de uma *Wireless Sensor Network* de teste através de aplicações desenvolvidas para o sistema operativo Contiki a funcionar sobre a plataforma μMRF.

4.1. A plataforma μMR

A plataforma na qual se pretende desenvolver uma solução para a integração do sistema operativo Contiki é denominada por μMRF e representa uma das soluções apresentadas pela empresa Micro I/O Serviços de Electrónica, Lda, para a área de desenvolvimento de soluções sem fio baseadas na norma IEEE 802.15.4.

Com um tamanho reduzido, menor que um cartão de crédito, um consumo moderado de energia e uma capacidade de processamento de até 40 MIPS esta plataforma apresenta-se como uma solução integrada para o desenvolvimento de WSNs.

A comunicação rádio da plataforma tem por base um *transceiver* rádio IEEE 802.15.4 da Microchip™, o MRF24J40MA, que torna possível operar nas bandas de rádio *industrial, scientific and medical* (ISM) não licenciadas, entre 2,4GHz e 2,48GHZ definidas pelo ITU-R.

Na Figura 4.1 encontra-se uma fotografia da plataforma para a qual é pretendido transpor o sistema operativo Contiki, assim como a indicação da localização dos principais componentes que a constituem.

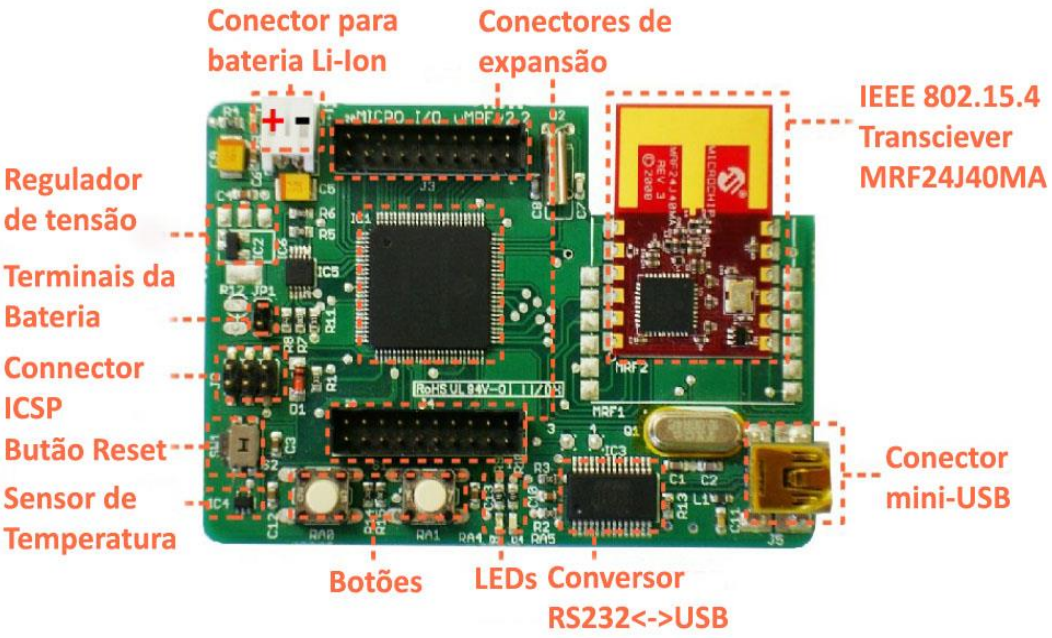

Figura 4.1: Plataforma *µMRF* [36].

As principais características da plataforma µMRF são:

- **MCU:** Com um microcontrolador dsPIC33FJ256MC710, desenvolvido pela Microchip™, baseado numa arquitetura Harvard modificada de 16-bit, com um poder computacional de até 40MIPs, a plataforma consegue interligar-se com o seu *transceiver wireless*, assim como a sensores e outras unidades externas graças à implementação em *hardware*, por parte do microcontrolador, de protocolos de comunicação, como são por exemplo SPI, UART, Enhanced CAN. O microcontrolador apresenta também três estados de funcionamento: RUN, IDLE e SLEEP, que, em conjunto com a sua capacidade de troca de relógio do sistema e variação da sua frequência, consegue obter mecanismos de poupança energética interessantes para este tipo de aplicações.
- **RAM:**30720 bytes
- **Memória Flash:** 256 Kbytes

- **Interface IEEE 802.15.4:** obtida através da solução MRF24J40MA da Microchip™. Esta apresenta uma antena em PCB integrada que através dos seus circuitos integrados e de uma interface SPI consegue interligar-se com os microcontroladores PIC® da Microchip™. Este *transceiver* rádio apresenta um baixo consumo energético (modo Rx tipicamente 19 mA, modo TX tipicamente 23 mA, e modo SLEEP tipicamente 2 µA) o que, aliado a um modo de funcionamento exclusivo, o modo Turbo, permite aumentar em duas vezes e meia a taxa de transmissão e receção de pacotes especificados segundo a norma IEEE 802.15.4 para os 625 kbps e o torna a solução ideal para o desenvolvimento de *Wireless Sensor Networks.*
- **Interface Mini-USB:** Apresenta um conector mini-USB integrado na plataforma, que, em conjunto com um conversor bidirecional RS232-USB, consegue fazer a interligação entre a interface UART do microcontrolador e o conector mini-USB, oferecendo, deste modo, a possibilidade de comunicar fisicamente com a plataforma.
- **Input/Output:** Os microcontroladores têm à disposição portas analógicas e digitais, assim como a possibilidade de gerar interrupções do sistema mediante alteração do estado de algumas das suas portas. Com todas as suas portas de entrada tolerantes a 5V, e com portas de saída com *output* máximo de 3,6V, esta plataforma apresenta as características ideais para que a interligação com dispositivos externos possa ser feita de forma simples e eficaz.
- **Conectores de Expansão:** A plataforma µMRF apresenta conectores de expansão para todos as portas não utilizadas do microcontrolador, permitindo assim que possam ser anexados à plataforma novos dispositivos de forma modular que, em conjunto com os protocolos de comunicação disponibilizados pelo microcontrolador, possam oferecer novas funcionalidades.
- **Sensor de temperatura:** Integrado na plataforma, temos um sensor de temperatura desenvolvido pela Microchip™, o MCP9700, que possibilita, através da variação da tensão do seu terminal de saída e da

implementação, oferecida pelo microcontrolador, de um conversor analógico-para-digital (conhecido pela sua sigla inglesa ADC), obter a temperatura ambiente com uma precisão de ±4ºC na gama de temperaturas entre 0ºC e 70ºC.

- **Alimentação energética:** A alimentação energética da plataforma pode ser feita de duas formas: através do seu conector mini-USB ou através do acoplamento de uma bateria Li-IoN. Caso se opte pelo uso de uma bateria como fonte de energia, esta pode ser recarregada através do conector mini-USB, existente na plataforma μMRF, pois esta tem incorporados os mecanismos de carregamento de bateria necessários.
- **LEDs e Botões:** A plataforma apresenta dois LEDs, um de cor verde e outro de cor laranja, que servem sobretudo como sinalizadores visuais que os programadores podem utilizar no desenvolvimento de aplicações. Apresenta também dois botões físicos que, para além das suas funções em aplicações, funcionam muitas vezes em conjunto com os LEDs da plataforma, como ferramentas de *debug* úteis.

4.2. Ferramentas de desenvolvimento utilizadas

Na elaboração de uma solução de integração do SO Contiki na plataforma μMRF foi utilizado o compilador C MPLAB® XC16, da empresa Microchip ™. Desta forma, recorreu-se a um compilador C especialmente desenvolvido pela Microchip ™ para a família de microcontroladores, que está em utilização na plataforma proposta para realizar a transposição do sistema operativo.

De forma a aproximar os futuros utilizadores desta plataforma, com uma solução de SO Contiki, com a utilização semelhante encontrada noutras plataformas que já têm desenvolvidas soluções com o SO Contiki, optou-se pela utilização de um editor de texto simples, o Notepad++, tendo sido feita posteriormente a integração dos diferentes ficheiros do sistema operativo e respetivas aplicações. Estas foram desenvolvidas através do uso de ficheiros especiais, as "*Makefiles"*, que contêm comandos *shell*, que visam compilar o sistema através de regras previamente

impostas para a correta integração dos diferentes diretórios e ficheiros necessários à implementação da imagem do sistema operativo durante o processo de compilação. O sistema operativo tem um ficheiro "Makefile.include" que é global a todo o sistema. Durante a transposição do sistema operativo para esta nova plataforma foram criados dois ficheiros "Makefiles" para associar os novos ficheiros e diretórios da nova plataforma e o novo microcontrolador ao ficheiro Makefile global do sistema operativo, o "Makefile.umrf" e "Makefile.dspic33fj256mc710", respetivamente.

A compilação do sistema operativo, e aplicações desenvolvidas, foi feita através de terminal UNIX, emulado no sistema operativo Windows 7, de 32bits, através do programa Cygwin, sendo, neste terminal, chamadas, através do comando Unix "make", as Makefiles desenvolvidas e o respetivo compilador de C, o MPLAB® XC16 da Microchip™.

4.3. Transposição do SO Contiki para a plataforma μMRF

O Sistema Operativo Contiki tem na sua génese uma implementação que visa simplificar a sua transposição para novas plataformas, apresentando para tal uma separação clara do código, que é completamente independente da plataforma para a qual o sistema será implementado, e o código que depende e necessita de adaptações. Desta forma, tentou fornecer-se aos programadores uma melhor perceção do que é necessário ser adaptado numa nova plataforma e do que é imutável nas diferentes soluções do sistema operativo para as diferentes plataformas já transpostas.

Para se efetuar a transposição do SO Contiki para uma nova plataforma, foi necessário analisar, primeiramente, como é que o mesmo está estruturado e, em seguida, analisar quais os módulos que necessitam de adaptação, consoante a plataforma física onde o sistema operativo irá estar em funcionamento. Inicialmente, optou-se por fazer uma breve análise do que é encontrado nos diferentes diretórios, existentes no diretório raiz do sistema operativo.Este último diretório está representado na Figura 4.2. De seguida, passou-se à identificação do que é necessário ser efetuado no âmbito deste trabalho.

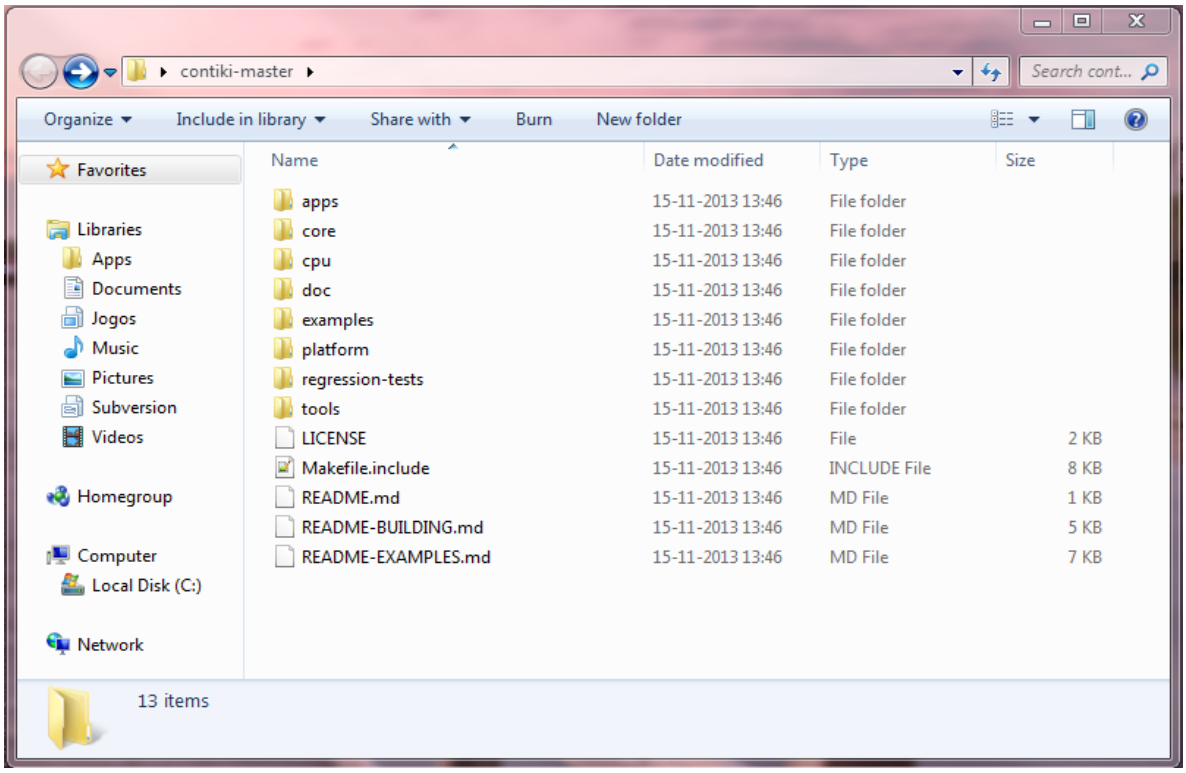

Figura 4.2: Directório raiz do SO Contiki.

No diretório raiz do sistema operativo, presente na Figura 4.2, estão presentes os seguintes diretórios:

- **"/apps":** Neste diretório estão disponíveis aplicações desenvolvidas para sistema operativo e que podem ser utilizadas no desenvolvimento de novas aplicações. São exemplo: a implementação de um "*webserver*" e uma linha de comandos *shell*, com a implementação de alguns comandos UNIX, entre outras aplicações que fornecem funcionalidades que podem ser incorporadas em aplicações criadas pelo programador.
- **"/core":** Neste diretório encontramos o *kernel* do SO, em conjunto com os seus mecanismos de gestão de processos e implementação de *protothreads*. Encontramos também as pilhas de comunicação implementadas no sistema operativo, com especial relevância para a pilha de comunicação µIPv6 que fornece protocolos de rede (protocolo IP) e de transporte (protocolo TCP, UDP), assim como a

possibilidade de utilizar, na camada de aplicação, o protocolo HTTP. Como mecanismo de adaptação de pacotes IPv6 sobre IEEE 802.15.4, utiliza-se a camada de adaptação desenvolvida pelo grupo IETF 6LowPan, também conhecida por esse nome [22].

- **"/cpu":** Neste diretório, estão definidos subdiretórios para cada um dos microcontroladores suportados pelo sistema operativo. No âmbito do desenvolvimento da transposição do sistema operativo Contiki para a plataforma *µ*MRF, foi criado um subdiretório específico para o novo microcontrolador, o "cpu/dspic33FJ256MC710".
- **"/doc":** Neste diretório, é possível encontrar especificações sobre o estilo de programação a ser utilizado no desenvolvimento de aplicações para o sistema operativo Contiki, nomeadamente no que diz respeito às indentações necessárias. Podemos também encontrar esqueletos de programas tipo, que nos permitem ter uma visão simplista de como criar aplicações, que serão posteriormente executadas sobre o sistema operativo.
- **"/exemples":** Neste diretório, estão patentes exemplos de aplicações já desenvolvidas para o sistema operativo Contiki que visam acelerar o processo de aprendizagem de criação e desenvolvimento de aplicações para Contiki. Apesar de ser redundante, uma vez que os exemplos criados no âmbito deste trabalho também são executáveis noutras plataformas que executem o sistema operativo (desde que tenham os mesmos componentes funcionais disponíveis), criou-se um subdiretório onde foram colocadas as aplicações exemplo, criadas durante este trabalho, o diretório "exemples\umrf".
- **"/platform":** Neste diretório, estão criados subdiretórios para todas as plataformas que se encontram adaptadas ao sistema operativo Contiki. No âmbito deste trabalho, foi criado um subdiretório específico, "platform\umrf", para a implantação do sistema operativo. Dentro desse subdiretório, foi posteriormente criado o "platform\umrf\dev", onde se colocaram os códigos fonte, correspondentes aos *device-*

drivers, criados e necessários, para o correto funcionamento de todas as funcionalidades da plataforma no sistema operativo.

- **"regression-tests":** De forma a assegurar que o código do sistema operativo Contiki funciona como esperado, foram recentemente desenvolvidos testes de regressão para algumas das plataformas que, através de simulações, conseguem observar como os mecanismos do SO Contiki funcionam. Estão já disponíveis para as principais plataformas transpostas.
- **"/tools":** Neste diretório, estão disponíveis ferramentas que facilitam a implementação de aplicações para o SO Contiki. No decorrer deste trabalho, foi utilizada a ferramenta "tunslip6", com vista a interligar a *Wireless Sensor Network* criada com a rede local do computador, através do envio de pacotes IPv6, via porta série, usando o protocolo SLIP [33].

Após esta breve análise sobre os conteúdos dos diferentes diretórios presentes no diretório raiz do sistema operativo Contiki, podemos observar que, no âmbito de uma transposição do sistema operativo para uma nova plataforma, dois diretórios tomam especial relevância, os diretórios "/platform" e "/cpu", que serão explorados de seguida.

4.3.1. Diretório "/platform"

Neste diretório, estão criados subdiretórios para todas as plataformas que suportam o SO Contiki. Dentro desses subdiretórios existem, pelo menos, três ficheiros essenciais, comuns a todas as plataformas com sistema operativo Contiki implementado: o "contiki-main.c", o "contiki-conf.h" e, por último, o "Makefile.platform", onde "platform" é o nome genérico, que é substituído pelo nome da plataforma.

Como foi anteriormente exposto, criou-se também um subdiretório, "platform/umrf/dev", onde foram adicionados os *device-drivers* necessários ao correto funcionamento dos diversos componentes da plataforma.

De seguida, vamos analisar os ficheiros criados no âmbito da transposição do sistema operativo Contiki para esta nova plataforma, dentro deste diretório

"contiki-main.c"

No ficheiro "contiki-main.c" está representada a função "main()" do sistema operativo para a plataforma em específico. É nesta função que são feitas as inicializações dos "drivers", processos e livrarias, necessárias ao correto funcionamento do SO Contiki, na respetiva plataforma.

Na presente transposição do sistema operativo Contiki para a plataforma μMRF, inicialmente, na função "main()", é chamada a função dspic33_init(), definida no ficheiro "dspic33.c", no diretório " /cpu/dspic33fj256mc710". Serve para configurar o modo de funcionamento do oscilador de quartzo que gerará o sinal de relógio, usado pelos *timers* do microcontrolador, com uma frequência de oscilação de 80 MHz. De seguida, através da função clock_init(), definida no ficheiro "clock.c", presente no diretório do microcontrolador da plataforma, é feita a configuração do *timer1* do microcontrolador, de modo a apresentar um período de geração de interrupções de aproximadamente 0,9766 ms, período este que será a menor unidade de tempo do sistema operativo na plataforma μMRF. A cada interrupção gerada pelo *timer1* do microcontrolador, irá incrementar-se a variável "count", que guardará o número de "ticks" passados desde a inicialização do sistema operativo.

A inicialização dos *drivers* responsáveis pela inicialização do *watchdog* do sistema operativo está também definida em dois passos, à semelhança do que aconteceu com a definição do relógio do sistema operativo. A principal função do *watchdog timer* é realizar um *reset* ao sistema, caso haja um problema de *software*, sendo também usado para colocar o microcontrolador num modo de funcionamento de baixo consumo energético, quando necessário. O *watchdog* do microcontrolador dsPIC33FJ256MC710 consiste na utilização do oscilador "low power RC oscillator" (LPRC) e de dois escalonadores, para definição do seu tempo de *time out*. O *watchdog* irá incrementar o seu valor segundo a frequência de funcionamento do

LPRC, até que seja atingido o seu valor de *time-out*, altura em que efetuará *reset* ao microcontrolador. De forma a não ocorrerem *resets* por *time-out* do *watchdog*, o sistema tem que realizar periodicamente uma operação de *clear* ao *watchdog*.

O primeiro passo da inicialização do *watchdog* é feito no ficheiro "dspic33.c", presente no diretório do microcontrolador, onde são feitas as configurações de inicialização do *watchdog* do microcontrolador, e definido o seu valor de *time-out*. O segundo passo da inicialização do *watchdog* realiza-se através da função "watchdog_init()", definida no ficheiro "watchdog.c", que se encontra no diretório do microcontrolador, e onde foram implementadas as funções de interligação entre o *watchdog* do microcontrolador e o sistema operativo.

De seguida, ainda na função "main()", é feita a inicialização da API do SO Contiki para gestão dos LEDs da plataforma, seguida da inicialização do modo de funcionamento pretendido para a porta série e das livrarias, que inicializam o funcionamento dos processos no sistema operativo. Estas inicializações são comuns à maioria das plataformas, com exceção para as diferentes livrarias que podem, por opção, não ser associadas à imagem do sistema. Pode-se assim obter uma imagem do sistema operativo de acordo com as necessidades da plataforma, podendo deste modo reduzir-se a utilização de memória disponível.

De forma a unificar o processo de inicialização das comunicações sem fios, foi criado na plataforma *µ*MRF um ficheiro chamado "init-net.c", presente no diretório da plataforma. Neste ficheiro, encontra-se definida uma função chamada "init_net()", que tem estabelecidos os passos necessários à inicialização do driver do *transceiver* rádio MRF24J40, assim como a definição do endereço MAC da plataforma e a atribuição de endereço IPv6, após inicialização da livraria TCP/IP disponível no SO Contiki. Esta última inicialização é efetuada nesta fase de execução da função "main()".

Após esta fase, são também inicializados os processos relativos aos diferentes *timers* do sistema operativo, como são exemplo: o "etimer", o "ctimer" e o "rtimer", assim como os processos que constam na lista "autostart_processes", através da chamada da função "autostart_start".

Após a execução desta fase de inicializações, o sistema entra num ciclo infinito característico do sistema operativo Contiki, que visa a chamada regular da

função “process_run()”. Caso a função “process_run()” retorne um valor igual a zero, isto é, não haja eventos pendentes nem processos a necessitar de *poll*, o microcontrolador entrará em modo de poupança de energia, sendo ativado mediante a ocorrência de uma interrupção externa.

O modelo de execução em ciclo infinito, encontrado após as primeiras inicializações, que se encontra estabelecido na função “main()”, está definido no espaço plataforma da Figura 4.3. Através da análise dessa figura podemos observar que, após a correta definição da função “main()”, e do relógio do sistema, torna-se possível ter funcional o sistema de escalonamento de eventos e de gestão de processos do sistema operativo Contiki.

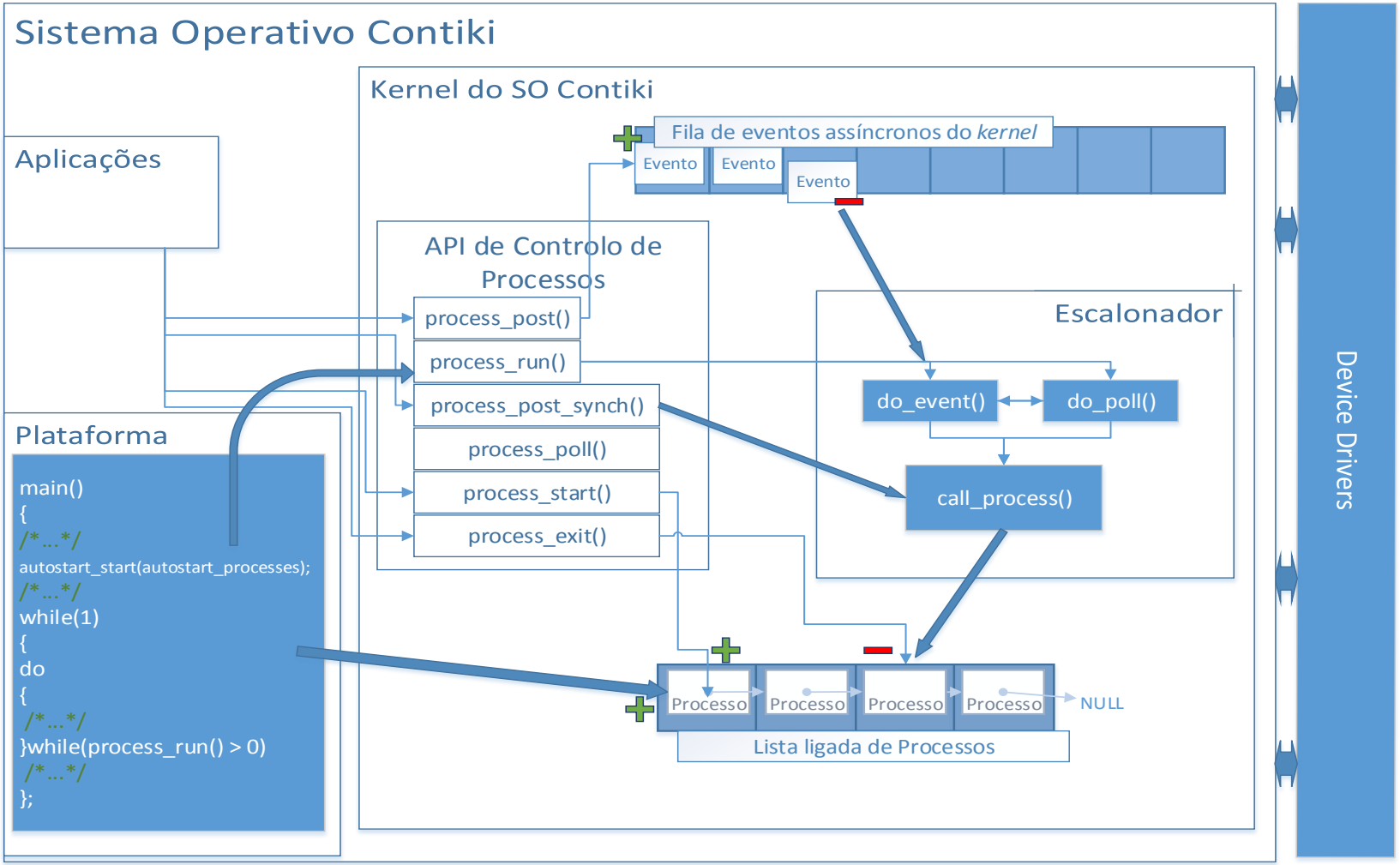

Figura 4.3: Visão global do funcionamento do SO Contiki.

“contiki-conf.h”

O ficheiro “contiki-conf.h” é o local definido no sistema operativo Contiki para se colocarem as configurações específicas para cada plataforma, assim como definir contantes e tipos de variáveis para que o sistema operativo seja executado corretamente.

No caso da plataforma µMRF, no “contiki-conf.h”, podemos encontrar:

- A inclusão das livrarias *standard* C “stdint.h” e “string”.
- Definidas variáveis com as frequências do cristal oscilador (MCU_FOSC) e frequência de funcionamento do microcontrolador (FCY).
- A definição dos tipos de variáveis que alguns módulos do sistema operativo utilizam, como seja a definição do tipo “rtimer_clock_t”.
- Definições necessárias para que haja comunicação na plataforma, quer a comunicação seja via porta série, quer seja via comunicação sem fios.

  Refira-se que, neste caso, é importante realçar os mecanismos de gestão de rede quando escolhemos ter uma *Wireless Sensor Network*, baseada em IPv6. Neste caso, foram escolhidos os seguintes *drivers* para configuração da pilha de rede:
  - NETSTACK_CONF_NETWORK: “sicslowpan_driver”
  - NETSTACK_CONF_FRAMER: “framer_802154”
  - NETSTACK_CONF_MAC: “nullmac_driver”
  - NETSTACK_CONF_RDC: “nullrdc_driver”
  - NETSTACK_CONF_RADIO: “mrf24j40_driver”
  - RIMEADDR_CONF_SIZE: 8
- Após escolhidos os *drivers* de configuração da pilha de rede, foram definidas configurações para as mesmas.Por exemplo, no mecanismo “siclowpan_driver” foi escolhido, como mecanismo de compressão de pacotes IPv6, o SICSLOWPAN_COMPRESSION_HC06 que se encontra definido no *standard* IETF, do grupo 6LowPAN, RFC 6282 [34].

“Makefile.µmrf”

Anteriormente tinha sido explicado que o SO Contiki tinha sido implementado com vista a simplificar a compilação do mesmo e das suas respetivas aplicações, nas diferentes plataformas suportadas, tendo para tal sido criado um ficheiro “Makefile.include” global que ficaria responsável por agregar todos os elementos necessários ao sistema operativo, assim como de incluir outros ficheiros “Makefile” correspondentes à implementação da plataforma e do microcontrolador utilizado. Este “ficheiro Makefile.include” encontra-se disponível na raiz do sistema operativo e, através da utilização do comando “make” e da indicação da plataforma para a qual é pretendido compilar uma aplicação, é iniciado o processo de compilação.

De forma a integrar esta nova plataforma no sistema operativo, o ficheiro “Makefile.umrf” teve de ser criado no diretório da plataforma em questão, o “platform/umrf/”. Neste ficheiro, estão discriminados quais os ficheiros da plataforma que devem ser adicionados ao ficheiro “Makefile.include”, disponível no diretório “./” do sistema, para posterior compilação. As indicações de qual o microcontrolador que a plataforma tem disponível e onde encontrar o seu respetivo ficheiro “Makefile”, têm de estar presentes no ficheiro “Makefile” da plataforma. É também usual encontrar no ficheiro “Makefile” a definição de macros condicionais, como, por exemplo, o número do nó da plataforma, para posterior definição aquando do pedido de compilação.

Este mecanismo de criação de macros condicionais, no ficheiro “Makefile” da plataforma, foi utilizado para obter diferentes endereços MAC consoante o valor de UMRF_ID. E ainda para ser possível selecionar o modo de funcionamento da porta série da plataforma, através do valor da macro SERIAL_LINE, ambos atribuídos durante o pedido de compilação por meio do comando “make” definido pela “Makefile” global do sistema operativo.

***Device-Drivers*:**

No decorrer da transposição do sistema operativo Contiki para a plataforma *µ*MRF surgiu a necessidade de desenvolver *device-drivers* com vista ao correto funcionamento dos dispositivos existentes na plataforma. Estes *device-drivers* foram alocados num subdiretório da plataforma, "platform/umrf/dev", e associados à compilação do sistema operativo através do ficheiro "Makefile" criado para a plataforma *µ*MRF.

Botões API

A implementação do uso dos botões para esta plataforma foi díspar do usualmente encontrado nas restantes plataformas já disponíveis com o sistema operativo Contiki.

Usualmente, a implementação encontrada para o uso de botões nas plataformas já disponíveis com o SO Contiki passa pela utilização da API para sensores, encontrada em "./core/lib/sensor.c", associada com a utilização de interrupções para deteção, se o botão for pressionado.

Os botões das plataformas têm de ser estabelecidos como estruturas do tipo "sensor_sensor", definidas na API para sensores do SO Contiki. E têm de ter a possibilidade de gerar interrupções para detetar trocas do estado quando pressionadas.

Quando um botão altera o seu estado, cria uma interrupção que irá ser gerida através da rotina de serviço à interrupção criada. Esta irá fazer um pedido de "*polling*" do processo "sensor_process" através da função "sensors_changed". O processo "sensor_process" é então escalonado. Após a alteração do estado do sensor na estrutura que o define, o processo "sensor_process" responsabiliza-se pelo envio de um evento do tipo "sensors_event", com um ponteiro para o botão, cujo estado se alterou, para todos os processos ativos no sistema operativo, de forma a informá-los da alteração do seu estado.

Este seria o procedimento a seguir para a gestão dos botões da plataforma μMRF, não fosse o facto de os botões desta não se encontrarem associados a portas do microcontrolador que suportem a utilização de interrupções.

Optou-se então por implementar um processo no sistema operativo Contiki, exclusivamente para verificar o estado dos dois botões existentes na plataforma, o "button_process". Este processo seria ativado de 250 ms em 250 ms, verificando aí o estado dos sensores. Quando fosse detetada uma transição do estado de um dos botões da plataforma, o processo faria um pedido de *polling* do processo "sensor_process", associado através da utilização da função "sensors_changed", indicando qual o botão que alterou o seu estado, à semelhança do que aconteceria numa rotina de serviço à interrupção, caso as interrupções pudessem ser usadas.

Desta forma, conseguiu-se usar o mecanismo de gestão de sensores já implementado no sistema operativo Contiki, sem usufruir do uso de interrupções, tendo, no entanto, consciência que tratando-se de um processo que é ativado de 250 ms em 250 ms, em situações de carga do sistema, pode prejudicar o seu desempenho.

LEDs API

A criação deste *device-driver* é fundamental para que a API para o LEDs do SO Contiki, criada com vista a uniformizar a sua utilização entre as diferentes plataformas, funcione. O *device-driver* criado encontra-se definido no ficheiro "leds-arch.c", disponível no diretório "./platform/umrf/dev/", e tem em si definidas as portas do microcontrolador às quais os leds da plataforma estão fisicamente ligados - uma variável estática responsável por guardar o estado dos leds da plataforma e três funções que servem posteriormente a LEDs API do SO Contiki: "leds_arch_init()", "leds_arch_get()", e "leds_arch_set()".

Na primeira função, "leds_arch_init()", são definidas as portas do microcontrolador que se encontram fisicamente ligadas aos leds da plataforma como saídas, e inicializadas com o estado lógico '0'. É também inicializada a variável que guarda o estado dos LEDs do sistema com o valor zero.

Na função "leds_arch_get()" é devolvido o valor guardado na variável "leds", variável que é utilizada para guardar o estado dos LEDs da plataforma.

Por último, na função "leds_arch_set(unsigned char l)" é criado o mecanismo de ativação dos leds da plataforma, mediante entrada de uma variável *unsigned char* que, mediante a definição de LEDS_GREEN, LEDS_RED ou LEDS_ALL, irá ligar o led verde, o led laranja ou ambos os LEDS, respetivamente, sendo atualizado o estado dos LEDs no sistema na variável leds.

No caso da plataforma μMRF, esta somente tem dois LEDs, um verde e um laranja, pelo que, na ausência de um parâmetro para o LED laranja, na API de LEDs do SO Contiki, foi criada, no ficheiro "platform/umrf/contiki-conf.h", uma definição para LEDS_ORANGE.

Sensor de temperatura

A plataforma μMRF tem embutido em si um sensor de temperatura desenvolvido pela Microchip™, o MCP9700. Este sensor converte a temperatura do ambiente que o rodeia numa diferença de potencial, que, mais tarde, após a conversão de analógico para digital, pode ser convertida para uma temperatura.

O sensor apresenta uma precisão de ±4ºC no intervalo de temperatura dos 0 ºC aos 70 ºC, sendo, desta forma, uma solução integrada viável para medições de temperatura.

O processo de desenvolvimento de um *device-driver* para este sensor passou pala configuração do conversor analógico para digital, disponível no microcontrolador. Foi definido um conversor analógico para digital com 12 bit disponíveis, que proporcionou um conversor analógico para digital de 4096 degraus possíveis o que, aliado à tensão de referência de 3,3 V, impôs uma resolução de 0,81 mV. No caso deste sensor de temperatura, pode ser excessivo, dada a sua gama de temperaturas, sendo no entanto importante a resolução apresentada na ADC em futuras utilizações da mesma. As configurações efetuadas podem ser observadas no ficheiro denominado "adc.c", que consta no diretório "\platform\umrf\dev\".

Através da análise do gráfico tensão de saída vs temperatura ambiente, que consta do *datasheet* disponibilizado pelo fabricante [37], verifica-se que, quando a temperatura ambiente se situa nos 0 ºC, o sensor de temperatura terá uma saída de 500 mV e que o sensor apresenta um declive de 10 mV/ºC. Com esta informação,

obteve-se uma equação de conversão da tensão de saída do sensor para graus Celsius, que se encontra definida na Equação 1.

$$T(°C) = \frac{\left(\left(V \times \frac{3300}{4096}\right) \times 500\right)}{10}$$

Equação 1 Equação de conversão Tensão para temperatura em ºC para sensor MCP9700.

De forma a tornar simples a utilização do sensor de temperatura, foi criado um ficheiro "temperature_sensor.c", no diretório "/platform/umrf/dev/", que tem somente implementadas duas funções "temp_init()" e "temp_value()". A primeira função fica responsável pela configuração do conversor analógico-digital do microcontrolador. A segunda função fica responsável pela obtenção do valor da temperatura em graus Celsius, através da utilização do conversor analógico- digital e da equação de conversão, Equação 1.

Transceiver Rádio

O *transceiver* rádio existente na plataforma *µ*MRF, o MRF24J40, já se encontra adaptado sob a forma de *radio-driver* do SO Contiki na plataforma "seedeye". Esta plataforma foi transposta recentemente pela *Scuola Superiore Sant'Anna* em Milão, Itália [38], pelo que a sua implementação na plataforma *µ*MRF passou, maioritariamente, pela integração do *radio-driver* já desenvolvido, com o microcontrolador dsPic33FJ256MC.

A integração entre o *radio-driver* do *transceiver* e o microcontrolador da plataforma *µ*MRF encontra-se no ficheiro "mrf24j40_arch.h", que se encontra disponível no diretório "/platform/umrf/dev/mrf24j40". Nele foi feito o mapeamento das portas utilizadas para interligar o microcontrolador com o *transceiver*, assim como foram definidas as funções de ativação das interrupções que serão geradas quando chegam pacotes ao *transceiver* rádio.

4.3.2. Diretório "/dspic33FJ256MC710"

Para cada novo CPU transposto para o SO Contiki tem de ser desenvolvido um ficheiro "clock.c" que ficará responsável pela definição da menor unidade de tempo do sistema operativo através da definição do período relógio do microcontrolador, e também pela implementação das funções definidas no ficheiro "clock.h", integrado no *core* do sistema operativo, no diretório "./core/sys". A implementação do sistema de gestão do relógio do SO Contiki, que será alocada no "clock.c", exige pelo menos estas três funções: "clock_init()", "clock_delay()" e "clock_time()".

Neste contexto de integração do microcontrolador dsPic33FJ256MC710 tiveram também de ser desenvolvidos os mecanismos de gestão da porta série, um *watchdog* para salvaguarda do funcionamento do sistema operativo e um ficheiro "Makefile" que garantisse a integração de todos os ficheiros criados para adaptar o sistema operativo ao microcontrolador, durante o processo de compilação global do sistema operativo.

Clock

O módulo "clock" é responsável pela interface de gestão da funcionalidade de relógio específico da plataforma e o SO Contiki. Este módulo tem uma função crucial para o correto funcionamento do sistema operativo pois está responsável pela medição do tempo do sistema, tendo para isso definida uma macro CLOCK_SECOND, que corresponde a um segundo, em "ticks" do relógio do sistema operativo. Um segundo nesta plataforma representa 1024 "ticks" de relógio, isto é, um segundo do sistema operativo corresponde a 1024 interrupções periódicas geradas devido às oscilações do cristal de quartzo existente na plataforma e às configurações impostas no *timer1* do microcontrolador. No caso da plataforma μMRF foi definido que a menor unidade de tempo, os "ticks" do sistema, tivesse uma resolução de aproximadamente 0,9766 ms.

O ficheiro "clock.c" encontra-se disponível no diretório "/cpu/dspic33FJ256MC710" e nele podemos encontrar, para além das três funções

essenciais, “clock_init()”, “clock_delay()” e “clock_time()”, a definição das seguintes funções:“clock_seconds()”, “clock_set_seconds()” e “clock_delay_usec()”.

Começando pela função clock_init(), esta está responsável por inicializar a livraria *clock* do SO Contiki e deve ser chamada na função “main()” do sistema operativo durante a sua inicialização, estando nela definida a frequência do *timer1* do microcontrolador. Este *timer* define a resolução mínima que o sistema operativo terá, isto é, o valor do “tick” do sistema operativo.

A função “clock_time()” está responsável por ler e retornar o valor global do tempo do sistema em “ticks” de sistema quando invocada. Sendo que a função “clock_seconds” tem as mesmas funcionalidades da “clock_time()”,devolvendo somente o tempo do sistema em segundos. Os valores globais do tempo do sistema operativo são guardados nas variáveis “count”, para os “ticks” do sistema”, e na variável “seconds”, para os segundos do sistema.

As funções “clock_delay()” e a “clock_delay_usec()” são responsáveis por fornecer a funcionalidade de criar atrasos no tempo de sistema, estando a primeira definida em segundos e a segunda, em milissegundos.

A cada oscilação do relógio é gerada uma interrupção no microcontrolador que é tratada por uma rotina de serviço à interrupção, definida no ficheiro “clock.c”. Este está presente no diretório do microcontrolador, que está responsável por incrementar a variável global do sistema, que conta o número de “ticks” do sistema operativo, verificar a existência de timers “etimer”, cujo tempo expirou e que necessitem de efectuar “poll” ao processo onde estão criados, e, a cada 1024 “ticks”, incrementar a variável “seconds” global do sistema operativo que representa o número de segundos.

Rtimer

Como foi explicado anteriormente no subcapítulo 3.2.1, na componente das Livrarias que diz respeito aos *Timers* implementados no SO Contiki, este tem um *timer* “rtimer” que implementa o módulo de gestão de escalonamento e execução de tarefas *real-time* com períodos de execução previsíveis presente no ficheiro “rtimer.c” no diretório “/core/sys/”.

No SO Contiki existem dois contextos de execução possíveis : o cooperativo e o preemptivo. No contexto cooperativo, os processos são executados de forma sequencial, de acordo com o seu escalonamento. Num contexto preemptivo, a ocorrência de uma tarefa de maior prioridade cria uma pausa temporariamente na tarefa atual e toma controlo do microcontrolador.

Os processos executam sempre de forma cooperativa. Somente se verificam preempções no caso de interrupções e de tarefas *real-time* [24]. Este conceito de dois contextos de escalonamento está representado na Figura 4.4.

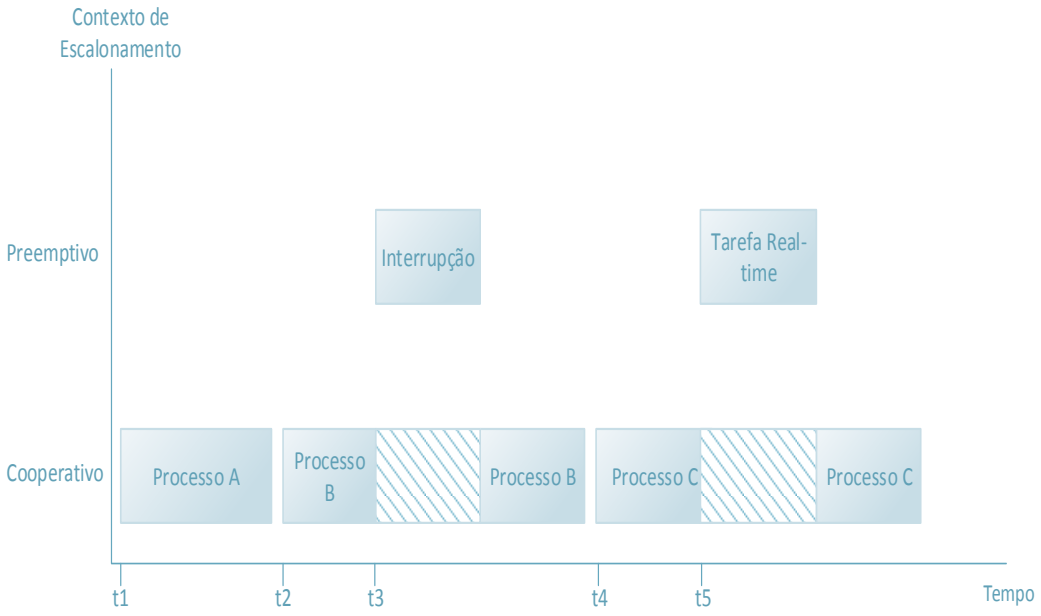

Figura 4.4: Contexto de escalonamento no SO Contiki [24].

Para que este mecanismo do sistema operativo funcione, foi definido, no diretório do microcontrolador, o ficheiro "rtimer-arch.c", que fica responsável por implementar as funções necessárias para a interligação do sistema de gestão Rtimer e o relógio do microcontrolador.

As funções implementadas são: "rtimer_arch_init()", "rtimer_arch_now" e "rtimer_arch_schedule".

Na primeira função está definido que o relógio da livraria Rtimer será interligado com o *Timer 2* do microcontrolador que, por sua vez, está associado ao oscilador do microcontrolador e que, após configuração, apresentará um período de 0,1 s.

O papel da segunda função é retornar o valor do relógio para o módulo de gestão de escalonamento "Rtimer", pelo que retorna o valor do registo do contador interno do *timer* 2 do microcontrolador.

A terceira função é responsável por ativar o *timer* 2 do microcontrolador, para o tempo que a tarefa *real-time* criada tem definido para ser executada.

UART e modos de operação da porta série

Para que o sistema operativo tenha um correto funcionamento da porta série, foi desenvolvido um *device-driver*, que se encontra-se definido em "serial.c" e disponível no diretório "/cpu/dspic33FJ256MC710/lib/". Aqui está implementada uma função de configuração da UART do microcontrolador, a "usartConfigBoard()", e funções de teste de leitura e escrita na porta série.

A função "usartConfigBoard()" é responsável por: configurar a UART1 do microcontrolador com um *baudrate* até 11520; estabelecer quando é que a mesma gera interrupções no microcontrolador e, ainda, por a ativar.

No ficheiro "serial.h" foi implementada a rotina de serviço à interrupção da UART1 do microcontrolador e, de forma a torná-la apta para os dois modelos de funcionamento do SO Contiki, utilizou-se um mecanismo de *callback* de funções que garante que, dependendo do modo de funcionamento (Serial Line ou SLIP), a função de gestão de receção de dados fosse alterada.

Para que os dois módulos de funcionamento da porta série pudessem ser utilizados, teve de implementar-se dois ficheiros que interligassem o *device-driver* criado com as livrarias Serial-Line e SLIP disponíveis no SO Contiki e referidas anteriormente no subcapítulo 3.2.1, em Serial-Line API e SLIP.

O ficheiro "serial-arch.c", presente no diretório "/cpu/ dspic33FJ256MC710/", irá fazer a interligação entre o *device-driver* da UART do microcontrolador e a livraria Serial-Line presente no SO Contiki. Este tem implementada uma função, "serial_line_setup()", que inicializa a UART1 do microcontrolador através da função "usartConfigBoard()", existente no *device-driver* criado, e a livraria Serial-Line do SO Contiki através da função "serial_line_init()" presente no ficheiro "core/dev/serial-line.h". No "serial-arch.c" está também indicada

a função “serial_line_input_byte”, pertencente à livraria Serial-Line do SO Contiki, como sendo a função que deve ser chamada na rotina de serviço à interrupção do UART para gestão dos dados a receber.

No ficheiro “slip-arch.c”, também ele presente no diretório do microcontrolador, temos definidos os mecanismos necessários à interligação da UART do microcontrolador e a livraria SLIP presente no SO Contiki.

Neste ficheiro são implementadas duas funções: “slip_arch_init()” e “slip_arch_writeb()”.

Na primeira função somente é feita a configuração da UART através da função “usartConfigBoard()”, presente no *device-driver* criado. Mais tarde efetua-se a inicialização do processo “slip_process”, na interface “slip-bridge.c”, definida no âmbito do *border-router* que será posteriormente utilizada na implementação de uma WSN de teste.

A segunda função é a que vai ser responsável pela gestão dos dados a transmitir que, neste caso específico, irá transmitir 1 byte pela porta série.

A função que fica responsável pela receção dos dados que chegam à UART do microcontrolador, e que deve ser chamada na rotina de serviço à interrupção da mesma, é a “slip_input_byte()” que se encontra disponível na livraria SLIP do SO Contiki, e em “/core/dev”.

Watchdog

A criação de um *watchdog* para o sistema operativo Contiki é imprescindível ao correto funcionamento do mesmo. Quando adequadamente implementado, este fornece mecanismos que, ao detetarem um bloqueio do sistema operativo, e após um tempo predefinido, forçam um *reset* do mesmo.

No SO Contiki, o *watchdog* é também usado para colocar o microcontrolador num modo de funcionamento de baixo consumo energético, quando não existem eventos a ser escalonados.

A livraria *watchdog*, encontrada no diretório “/core/dev” do sistema operativo Contiki, impõe que esta seja constituída pelas funções que encontramos na Tabela 4.1.

Tabela 4.1: Funções disponíveis no módulo watchdog.

Função	Descrição
watchdog_init ()	**Responsável por fazer reset à IDLE flag e WDT time out flag, e ao Software disable WDT bit**
watchdog_start ()	**Inicializa o *watchdog timer* do microcontrolador, através da activação do "Software disable WDT bit"**
watchdog_periodic ()	**É feito um "clear" ao WDT, ao IDLE flag , ao WDT Time-out-flag bit, e é iniciaçizado o WdT através da activação do "software disable WDT bit"**
watchdog_stop	**Desabilita o funcionamento do WDT através da atribuição do estado lógico 'o' ao "Software Disable WDT bit"**
watchdog_reboot	**Reinicializa o sistema operativo.**

As implementações destas funções encontram-se no diretório pertencente ao microcontrolador, no ficheiro "watchdog.c". Este ficheiro, em conjunto com o ficheiro "dspic33.c", assegura as configurações necessárias para que o dspic33FJ256MC710 tenha o seu *watchdog* interno ativado, assim como os mecanismos que impõem a entrada e saída do microcontrolador em modo IDLE bem definidas.

Makefile.dspic33FJ256MC710

Como ainda não existe nenhuma implementação do sistema operativo Contiki para o microcontrolador da Microchip™ dspic33FJ256MC710, a sua integração necessita de um ficheiro "*Makefile.dspic33fj256mc710*", que assegure que os ficheiros, necessários ao funcionamento do microcontrolador, estejam identificados, para posterior compilação com o sistema operativo, assim como estejam presentes as opções de compilação específicas do microcontrolador.

4.4. Implementação da *Wireless Sensor Network*

Com vista à implementação de uma *Wireless Sensor Network* demonstrativa do funcionamento do sistema operativo, foram criadas soluções que fornecessem funções diferentes aos nós que compõem a rede.

Ao nível da camada de acesso, o sistema operativo foi configurado em todos os nós da rede para que o seu *transceiver wireless* tenha ativo: o módulo NULLMAC para que, ao nível à camada MAC, não seja imposta qualquer condição de acesso; e um módulo NULLRDC para que seja imposto um *radio duty cycling* (RDC) que garanta que o *transceiver wireless* se mantém sempre ligado.

O sistema operativo Contiki proporciona implementações ao nível da camada de acesso, como por exemplo o ContikiMAC, que fornece um mecanismo de gestão eficiente dos ciclos de funcionamento do *transceiver* rádio, permitindo assim que seja feita uma otimização da energia utilizada [39]. No entanto, devido à fase inicial desta implementação, optou-se pelo não uso deste módulo, simplificando assim a avaliação do funcionamento da rede criada.

Ao nível da camada de Rede e de Transporte, uma vez que as plataformas detêm um *transceiver* baseado em IEEE 802.15.4, foi utilizada a implementação do mecanismo de compressão de cabeçalhos 6LoWPAN, de forma a poder ser usado, na camada de rede, o protocolo IPv6, implementado pela pilha de comunicação µIPv6, desenvolvida pela Cisco para o SO Contiki [25]. O mecanismo de compressão de cabeçalhos 6LoWPAN, utilizado nos nós da rede formada, encontra-se definido no *standard* IETF RFC 6282 [34], disponibilizado na solução 6LoWPAN, disponível no SO Contiki.

A camada de rede foi configurada com os protocolos UDP e ICMPv6, e como protocolo de *routing* foi configurado o "*routing protocol for low-power lossy IPv6 networks*" (RPL), cujas especificações são apresentadas no *standard* IETF RFC6550 [40].

De forma a ser possível aceder à rede de sensores criada através da Internet, é necessário que um dos nós da rede seja responsável pela tarefa de interligar as redes. Para tal, foi atribuído a um dos nós da rede a responsabilidade de ser o *border-router* entre as redes de sensores e a rede Internet IPv6. Esta interligação entre redes é feita através de uma plataforma µMRF, utilizando a solução *border-router,* disponibilizada

no sistema operativo Contiki em "contiki/exemples/ipv6/ rpl-border-router/", e uma estação base, que receberá os pacotes IPv6 do *border-router*, através da sua porta USB, usando o protocolo SLIP [33]. A estação base é um computador pessoal que usa o sistema operativo Linux Ubuntu [41] e que, através da ferramenta "tunslip6" disponível no diretório "contiki/tools/", criará uma interface "tun" que dará à plataforma conectada o endereço aaaa:1 na rede local. Esta receberá os pacotes IPv6 enviados pelo *border-router* o que permite transmiti-los para aplicações do utilizador para um posterior uso e/ou análise [42].

Na Figura 4.5, apresenta-se um esquema do funcionamento do mecanismo implementado para interligar a *wireless sensor network* criada com a rede IpV6 local da estação base, descrito anteriormente.

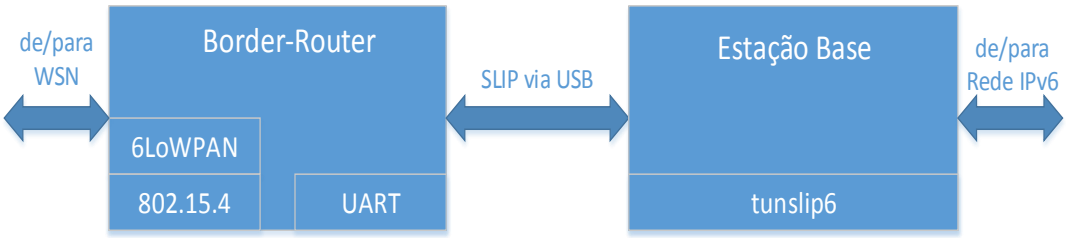

Figura 4.5: Funcionamento do *border-router*.

Os restantes nós que formam a rede, chamados "nós sensoriais", devido às suas funções, foram implementados de forma a fornecerem uma perspetiva das funcionalidades que este tipo de redes pode ter.

Neste caso particular, foi desenvolvida uma aplicação responsável por efetuar medições da temperatura do ambiente envolvente à plataforma e, através da API de criação de gráficos, desenvolvida e disponibilizada pela Google [43], foi criada uma página HTML que expusesse esses mesmos valores sob a forma de gráfico. Na Figura 4.6, expõe-se a página HTML, obtida num *browser* da estação base, através do IPv6 de um dos nós sensoriais da rede de teste que foi desenvolvida.

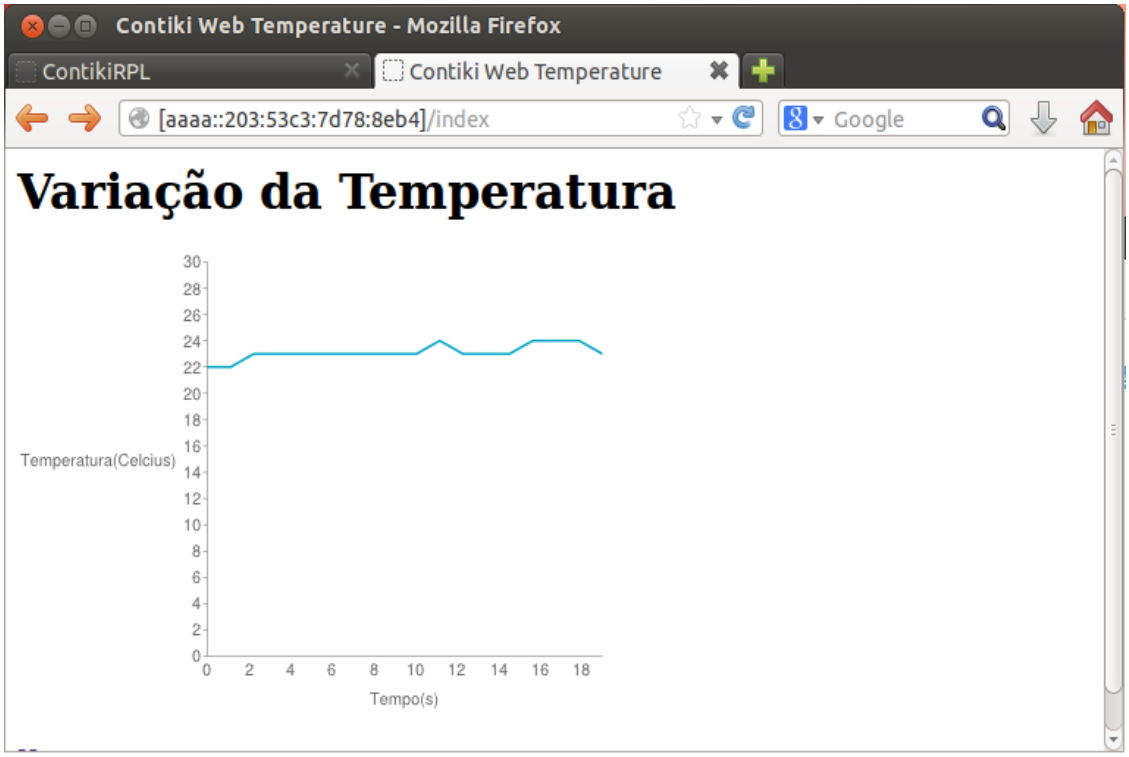

Figura 4.6: Página HTP obtida através do IPv6 identificador do nó sensorial.

5. Avaliação

Neste capítulo, descreve-se a *Wireless Sensor Network* referida no final do subcapítulo 4.4 para efeitos de testes ao funcionamento do sistema operativo Contiki na plataforma μMRF. Sobre ela foram realizados testes funcionais à pilha de comunicação IPv6 existente.

5.1. Sistema de testes

A WSN que serviu de "*testbed*" consiste em três plataformas *μ*MRF, duas delas implementadas com a aplicação feita através de HTML, para fornecer um gráfico das temperaturas envolventes à plataforma, tendo estas ficado conhecidas por "nós sensoriais", e uma terceira plataforma a servir de *border-router* entre a rede formada pelas plataformas *μ*MRF e a rede local da estação base.

Na Figura 5.1 temos uma representação esquemática da rede experimental desenvolvida, sendo esta constituída pelos seguintes elementos:

1 *Border-Router*

2 Nós Sensoriais

1 Estação base

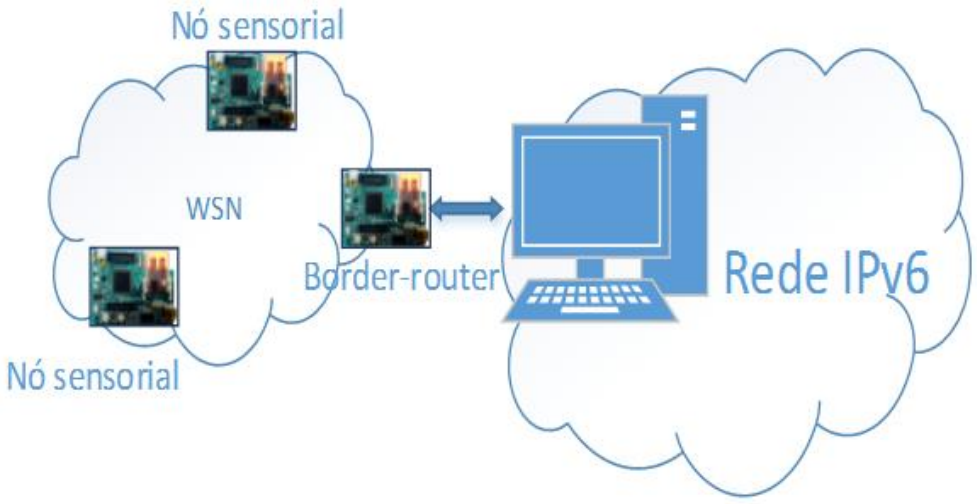

Figura 5.1: Rede experimental criada para fins de demosntração.

5.2. Metodologia e análise de performance de rede

A metodologia de análise das reais capacidades de rede que a WSN formada tem, foi feita com base nas métricas *Round Trip Time* e *Packet Loss*, obtidas através de pedidos de "ping", efetuados na estação base, para os IPs específicos dos nós sensoriais que formam a WSN.

- **Round Trip Time (RTT)**

  Em Telecomunicações, *Round-Trip Time* (RTT) é o intervalo de tempo entre o envio de um pedido de eco e a chegada de uma resposta de confirmação. No contexto das redes computacionais, este intervalo de tempo consiste na soma do tempo de envio e retorno de pacotes, também conhecido por tempo de *ping*.

- **Packet Loss (PLOSS)**

  Ocorre quando um ou mais pacotes não alcançam o seu destino na rede. As causas mais comuns para a existência de perda de pacotes em redes passam por: degradação de sinal no meio físico; existência de congestão do canal; problemas de *hardware* na rede; capacidades limitadas de armazenamento de pacotes por parte dos *transceivers* na camada física e MAC

Para realizarmos testes a estas métricas precisamos primeiro de saber os IPs das plataformas que representam os nós sensoriais da nossa WSN de teste. Ao usarmos a *tool* "tunslip6", fornecida no diretório "./tools", através do comando "sudo ./tunslip6 –s /dev/ttyUSB0 aaaa::1/64", conseguimos interligar a WSN criada com a rede IpV6 local da estação base, aparecendo no terminal a informação do IP da plataforma que está a servir de *border-router*. Ao acedermos, através de um *browser*, a esse mesmo IP, temos acesso à tabela de *routing* do *border-router*. Aí, podemos obter os IPs dos nós sensoriais da nossa rede. Na Figura 5.2, apresenta-se o *output* obtido ao acedermos ao IP da plataforma que está a servir de *border-router* entre a WSN de teste e a estação base.

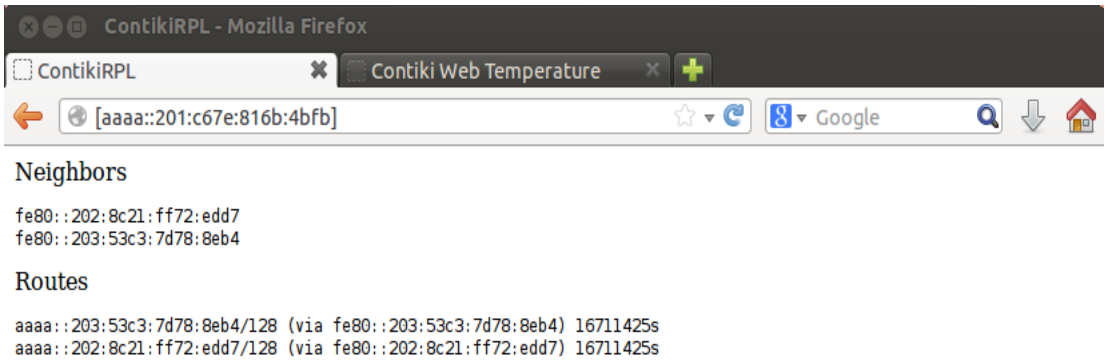

Figura 5.2: Tabela de routing obtida através do IP da plataforma bourder-router.

Após a obtenção dos IPs dos nós sensoriais da nossa rede sensorial, e de forma a avaliar as métricas *round trip time* e *packet loss*, foi realizado, na estação base, um teste que consistiu em efetuar 500 pedidos de *ping* para cada variação do *payload* do pacote IPv6. Utilizou-se para isso o comando *ping6*, que usa o protocolo Internet Control Message Protocol 6 (ICMPv6), com o endereço IPv6 de um dos nós sensoriais. Desta forma, foi possível testar a conectividade entre a estação base e o nó sensorial da WSN, tendo-se obtido os gráficos presentes na Figura 5.3 e Figura 5.4 [44].

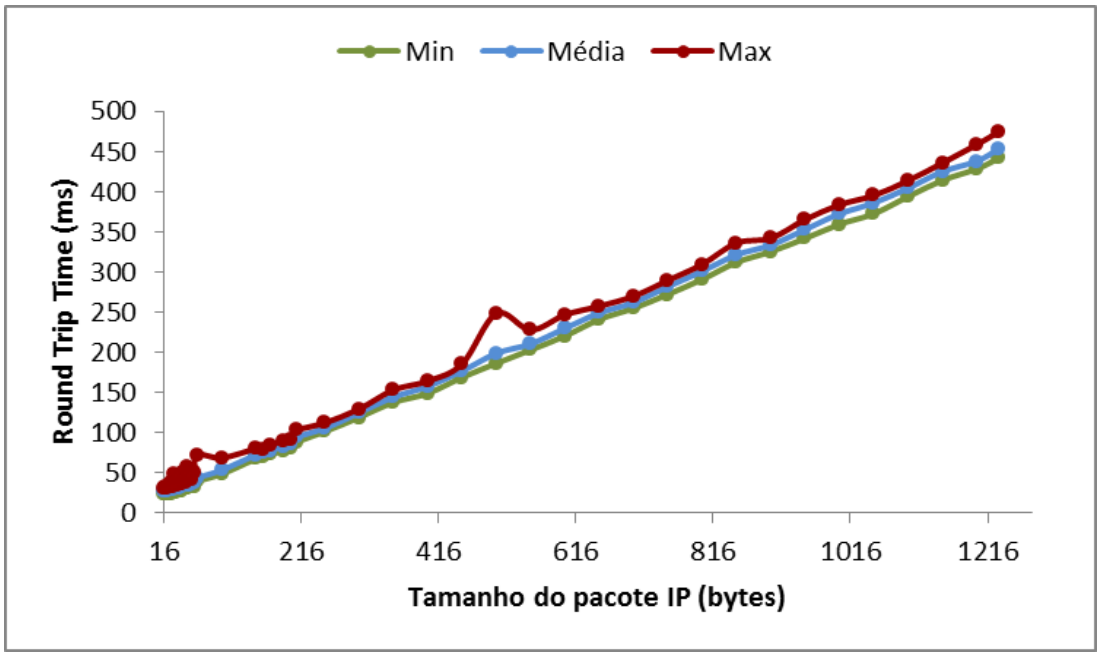

Figura 5.3: Evolução do Round Trip Time de acordo com o tamanho do payload do pacote ICMPv6.

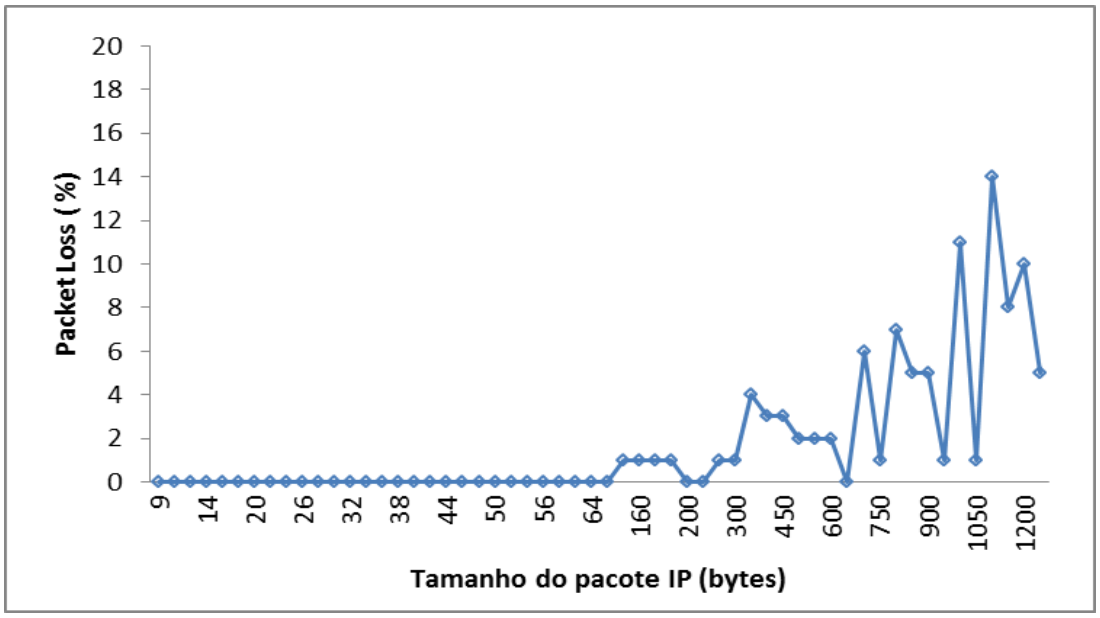

Figura 5.4: Evolução do Packet Loss de acordo com o tamanho do payload do pacote ICMPv6.

Nas Figura 5.3 e Figura 5.4, podemos observar a evolução do *Round Trip Time* e *Packet Loss* com o aumento do tamanho do pacote IP enviado. A Figura 5.4 mostra um aumento dos pacotes perdidos quando o tamanho do pacote ICMP ultrapassa os 64 bytes, comportamento explicado pelo mecanismo de fragmentação de pacotes IP, definido pelo mecanismo de encapsulamento e compressão de cabeçalhos, que permite o envio e receção de pacotes IPv6 em redes baseadas em IEEE 802.15.4, desenvolvido pelo grupo IETF "IPv6 over Low Power Wireless Personal Area Networks" (6LoWPAN) [22].

Apesar do aumento de pacotes perdidos quando os pacotes IPv6 têm um *payload* superior a 64 bytes, as Figura 5.3 e Figura 5.4 mostram que existe conectividade entre a rede da estação base e a WSN e que a pilha de comunicação 6LoWPAN, implementada no sistema operativo Contiki, funciona com os valores experimentais de RTT e PLOSS, que tornam possíveis o seu uso para aplicações, usando IPv6 sobre IEEE 802.15.4.

6. Conclusões e trabalhos futuros

Neste trabalho propôs-se fazer a transposição do sistema operativo Contiki para a plataforma μMRF desenvolvida pela Micro I/O Serviços de Eletrónica Lda. Esta plataforma contém um microcontrolador para o qual este sistema operativo ainda não se encontrava disponível. Adicionalmente, efetuou-se a adaptação desse sistema para permitir incorporar o *transceiver* MRF24J40MA, permitindo assim o estabelecimento de redes segundo a norma IEEE 802.15.4. Dotou-se ainda o sistema de acesso a um sensor de temperatura MCP9700. Refira-se que todos estes dispositivos foram desenvolvidos e são disponibilizados no mercado pela empresa Microchip™.

Desta experiência verificou-se que, apesar da pouca documentação existente sobre o processo de transposição do SO Contiki para uma nova plataforma, foi conseguida uma implantação funcional do mesmo na plataforma μMRF e que a mesma apresenta alguma robustez, no que diz respeito às comunicações *wireless* e à gestão de processos do sistema operativo.

Através do desenvolvimento de aplicações de teste, verificou-se que, com o sistema operativo Contiki, é de facto possível existir uma maior aproximação do conceito de IoT (internet das coisas, *Internet of Things*), onde todos os dispositivos passam a ser identificados por um endereço IPv6 único. Verificou-se também que as implementações de *websites*, via protocolo http, se tornam-se triviais, deixando o utilizador livre de responsabilidades na implementação das pilhas de comunicação, estando unicamente focado no desenvolvimento de aplicações.

Em suma, conseguiu-se transpor o sistema operativo para a plataforma disponibilizada, e constataram-se as vantagens da sua utilização para futuros trabalhos na área de WSNs.

A documentação deste trabalho fica também como um registo mais detalhado sobre o funcionamento do SO Contiki, assim como uma indicação exaustiva de quais os passos necessários para que seja feita a transposição do sistema operativo para uma nova plataforma.

Refira-se que este trabalho constitui uma parte importante para a simplificação da implementação de aplicações sobre a plataforma μMRF, estando já

em vista a sua utilização num subsistema de deteção de lugares livres ou ocupados, parte de um sistema para estacionamento inteligente (“smart parking”).

6.1. Trabalhos futuros

No que diz respeito a trabalhos futuros seria interessante que a transposição da funcionalidade do sistema operativo de reprogramação via comunicação sem fios fosse implementada na plataforma μMRF. De facto, devido às limitações temporais e à utilização de *bootloader* na memória da plataforma para programação via comunicação série, torna-se morosa a programação de módulos de uma WSN. Por isso, é prevista como tarefa para um trabalho futuro a implementação desta funcionalidade que, sem dúvida, aumentará o interesse na plataforma μMRF para desenvolvimento de aplicações em WSN.

Seria também interessante explorar de forma mais intensa os mecanismos de rede oferecidos pela pilha de comunicação IPv6 e de outros protocolos da camada de aplicação do modelo OSI também disponíveis no sistema operativo como é o caso do Constrained Application Protocol (CoAP) desenvolvido pelo grupo IETF CoRE Working Group [45] e desenhado para dispositivos IoT e Machine-2-Machine, que certamente abrirão portas a novas possibilidades de aplicações.

Referências Bibliográficas

[1] J. D. Kenney, D. R. Poole, G. C. Willden, B. A. Abbott, A. P. Morris, R. N. McGinnis, and D. A. Ferrill, "Precise positioning with wireless sensor nodes: Monitoring natural hazards in all terrains," in *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, 2009, pp. 722-727.

[2] C. Wei, X. Ge, E. Yaprak, R. Lockhart, Y. Taiqian, and G. Yanqing, "Using Wireless Sensor Networking (WSN) to Manage Micro-Climate in Greenhouse," in *Mechtronic and Embedded Systems and Applications, 2008. MESA 2008. IEEE/ASME International Conference on*, 2008, pp. 636-641.

[3] D. Wei, C. Chun, L. Xue, and B. Jiajun, "Providing OS Support for Wireless Sensor Networks: Challenges and Approaches," *Communications Surveys & Tutorials, IEEE*, vol. 12, pp. 519-530, 2010.

[4] R. v. Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," presented at the Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9, Lihue, Hawaii, 2003.

[5] S. Mishra and R. Yang, "Thread-based vs event-based implementation of a group communication service," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, 1998, pp. 398-402.

[6] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, 2004, pp. 455-462.

[7] 07-01-2013). Atmel. Available: http://www.atmel.com/

[8] (07-01-2013). *Microchip*. Available: http://www.microchip.com/

[9] (07/01/2013). *Texas Instruments*. Available: http://www.ti.com/mcu/docs/mculuminaryprodsearch.tsp?sectionId=95&tabId=2485&familyId=1755

[10] K. Romer and F. Mattern, "The design space of wireless sensor networks," *Wireless Communications, IEEE*, vol. 11, pp. 54-61, 2004.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGPLAN Not.*, vol. 35, pp. 93-104, 2000.

[12] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," presented at the Proceedings of the 2nd international conference on Embedded networked sensor systems, Baltimore, MD, USA, 2004.

[13] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler, "The *nesC* language: A holistic approach to networked embedded systems," presented at the Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, San Diego, California, USA, 2003.

[14] A. Dunkels, "Full TCP/IP for 8-bit architectures," presented at the Proceedings of the 1st international conference on Mobile systems, applications and services, San Francisco, California, 2003.

[15] C. A. a. t. S. I. o. C. S. (SICS). (2008, 08-01-2013). *IPv6-ready protocol stack*. Available: http://newsroom.cisco.com/dlls/2008/prod_101408e.html

[16] A. Dunkels, *Rime - A Lightweight Layered Communication Stack for Sensor Networks*, 2007.

[17] C. Thang Vu, C. Hung Nguyen, and H. Thanh Nguyen, "A comparative study on operating system for Wireless Sensor Networks," in *Advanced Computer Science and Information System (ICACSIS), 2011 International Conference on*, 2011, pp. 73-78.

[18] (08-01-2013). *Networked and Embedded Systems Lab(NESL)* Available: http://nesl.ee.ucla.edu/

[19] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," presented at the Proceedings of the 3rd international conference on Mobile systems, applications, and services, Seattle, Washington, 2005.

[20] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms," *Mob. Netw. Appl.*, vol. 10, pp. 563-579, 2005.

[21] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks," presented at the Proceedings of the 7th international conference on Information processing in sensor networks, 2008.

[22] IETF. (31/10/2013). *IPv6 over Low power WPAN (Active WG)*. Available: http://tools.ietf.org/wg/6lowpan/

[23] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," presented at the Proceedings of the 4th international conference on Embedded networked sensor systems, Boulder, Colorado, USA, 2006.

[24] A. Dunkels. (2013, 8-01-2013). *Details on Contiki's organization and subsystems: Processes* Available: https://github.com/contiki-os/contiki/wiki/Processes

[25] (2013, 20-10-2013). *Contiki: The Open Source OS for the Internet of Things*. Available: www.contiki-os.com

[26] A. Dunkels. (2013, 8-01-2013). *Details on Contiki's organization and subsystems: Memory allocation* Available: https://github.com/contiki-os/contiki/wiki/Memory-allocation

[27] N. Tsiftes, A. Dunkels, H. Zhitao, and T. Voigt, "Enabling large-scale storage in sensor networks with the Coffee file system," in *Information Processing in Sensor Networks, 2009. IPSN 2009. International Conference on*, 2009, pp. 349-360.

[28] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," presented at the Proceedings of the thirteenth ACM symposium on Operating systems principles, Pacific Grove, California, USA, 1991.

[29] P. Seebach. (2013, 08-01-2013). *Standards and specs: An unsung hero: The hardworking ELF*. Available: http://www.ibm.com/developerworks/power/library/pa-spec12/

[30] A. Dunkels. (2013, 08-01-2013). *Details on Contiki's organization and subsystems: Dynamic Loader*. Available: https://github.com/contiki-os/contiki/wiki/The-dynamic-loader

[31] A. Dunkels. (2013, 8-01-2013). *Details on Contiki's organization and subsystems: Timers*. Available: https://github.com/contiki-os/contiki/wiki/Timers

[32] A. Dunkels. (2013, 08-01-2013). *Details on Contiki's organization and subsystems: Input and Output*. Available: https://github.com/contiki-os/contiki/wiki/Input-and-output

[33] I. E. T. F. (IETF), "A nonstandaard for transmission of IP datagrams over serial lines: SLIP," ed, 1988.

[34] I. E. T. F. (IETF), " Compression Format for IPv6 Datagragrams over IEEE 802.15.4-Based Networks," ed, 2011.

[35] A. D. a. F. Ö. a. Z. He, "An adaptive communication architecture for wireless sensor networks," in *In Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems (SenSys 2007)*, Sydney, Australia, 2007.

[36] "uMRF: an extensible wireless board," L. Micro I/O serviços electrónica, Ed., 0.1 ed, 2011.

[37] Microchip™, "MCP9700/01- Low Power Linear Active Thermistors ICs," ed, 2009.

[38] G. Pellerano. Contiki SeedEye Platform project [Online]. Available: https://github.com/contiki-os/contiki/tree/master/platform/seedeye

[39] A. Dunkels, "The ContikiMAC Radio Duty Cycling Protocol " December 2011.

[40] I. E. T. F. (IETF), "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," ed, 2012.

[41] (30/10/2013). *Ubuntu Operating System*. Available: http://www.ubuntu.com

[42] M. Krasnyansky. (2000, 30/10/2013). *Universal TUN/TAP device driver*. Available: https://www.kernel.org/doc/Documentation/networking/tuntap.txt

[43] G. Inc. (30/10/2013). ***Google Charts***. Available: https://developers.google.com/chart/?hl=pt

[44] IETF, " Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," vol. Request for Comments: 4443 ed.

[45] IETF. (31/10/2013). *CoRE Working Group (Active WG)*. Available: http://tools.ietf.org/wg/core/